Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 5: this Friday @ 11:59 pm
  - Lab 3: Tuesday 2/14 @ 11:59 pm
  - Assignment 1: Friday 2/17 @ 11:59 pm

Previous lecture

- Digital Searching Problem
  - What if we use the fact that data items are represented as bits in computer memory?
- Digital Search Tree (DST)

This Lecture

- Digital Search Tree (DST)
- Radix Search Trie (RST)
- De La Briandais (DLB) Trie

Digital Search Trees (DSTs)

Instead of looking at less than/greater than, lets go left or right based on the bits of the key

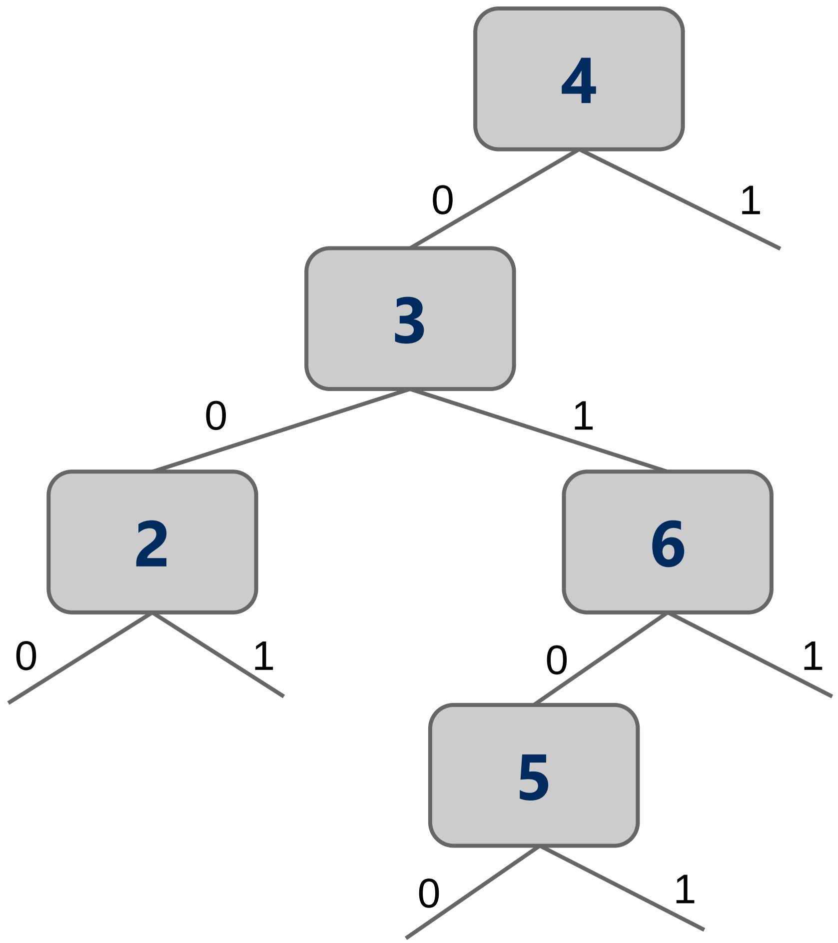
DST example: Insert and Search

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

Search:

3	0011
7	0111



Inserting into a DST

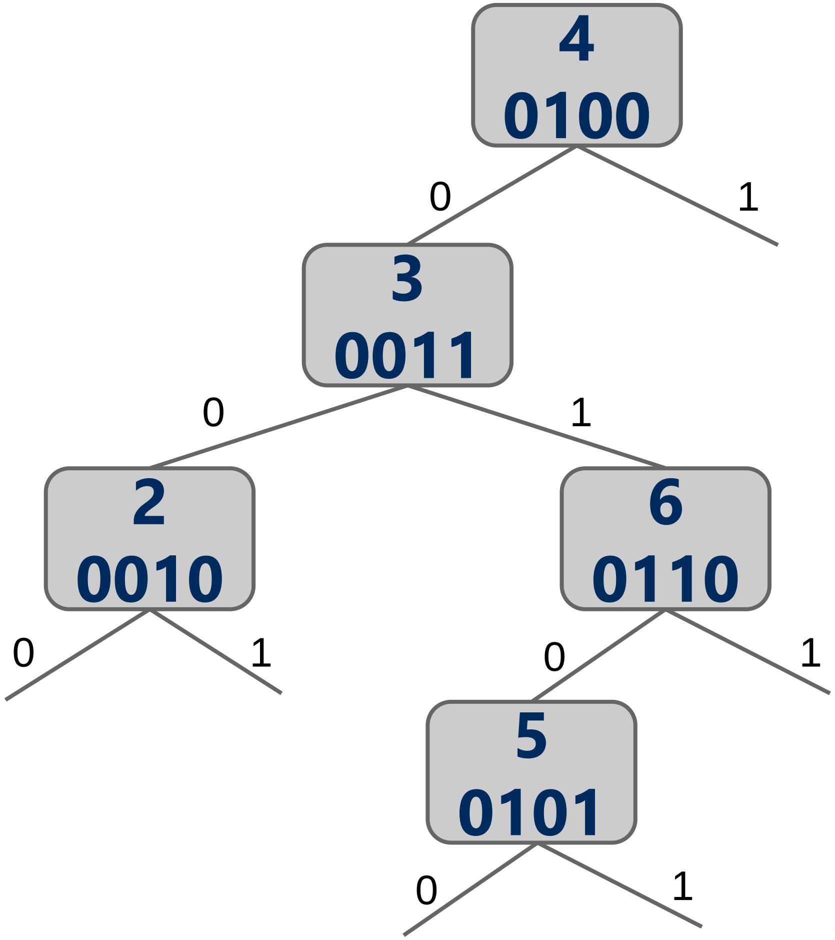
- adding a key *k* and a corresponding value
  - if root is null, add *k* as the root and return
  - current ← root
  - Repeat
    - if k is **equal to** the current.key, replace value and return
    - if current bit of k is 0,
      - if left child is null, add *k* as left child
      - else continue to left child (recursive call or current ← current.left)
    - if current bit of k is 1,
      - if left child is null, add *k* as right child
      - else continue to right child (recursive or current ← current.right)
- When does the algorithm stop?
  - no more bits or
  - hitting a null

Runtime Analysis of Digital Search Trees

- b: the bit length of the target or inserted key
- n: number of nodes in the tree
- Worst-case Runtime?
  - min(b, height of the tree)
- What is the average height of the tree?
  - Assume that having 0 or 1 is equally likely at each bit
  - log(n)
  - In general b >= $\lceil \log n \rceil$

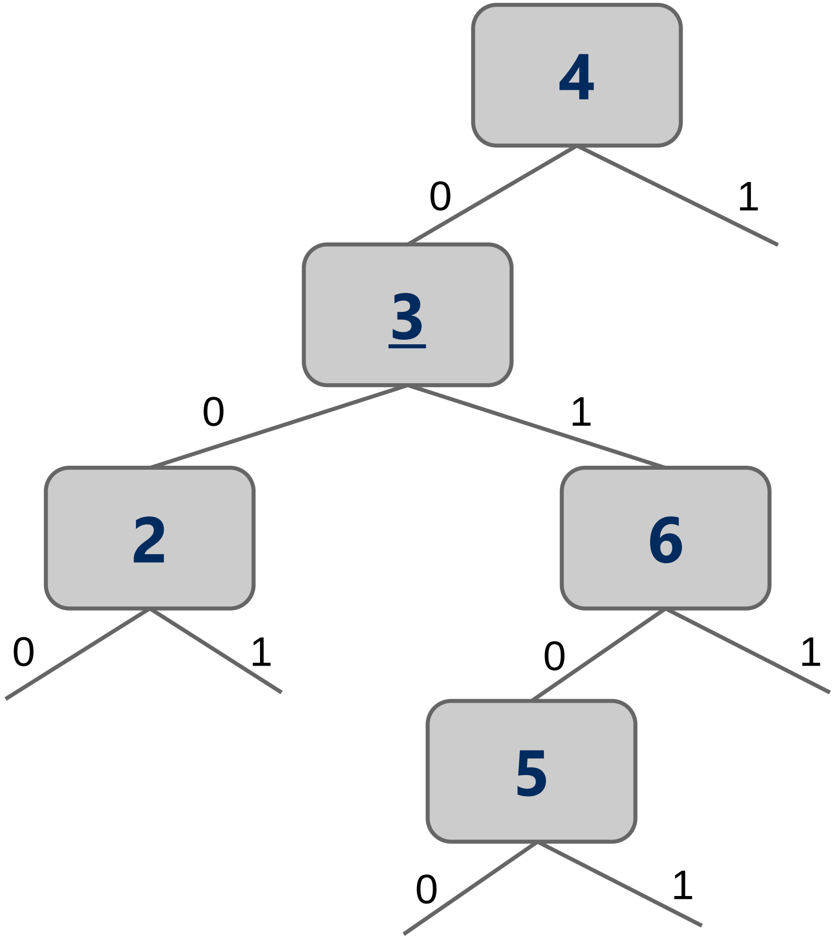
What property does a DST hold?

- In a DST, each node shares a **common prefix of length depth(node)** with all nodes in its subtree
  - e.g., 6 shares the prefix “01” with 5
- In-order traversal doesn’t produce a sorted order of the items
  - Insertion algorithm can be modified to make a DST a BST at the same time



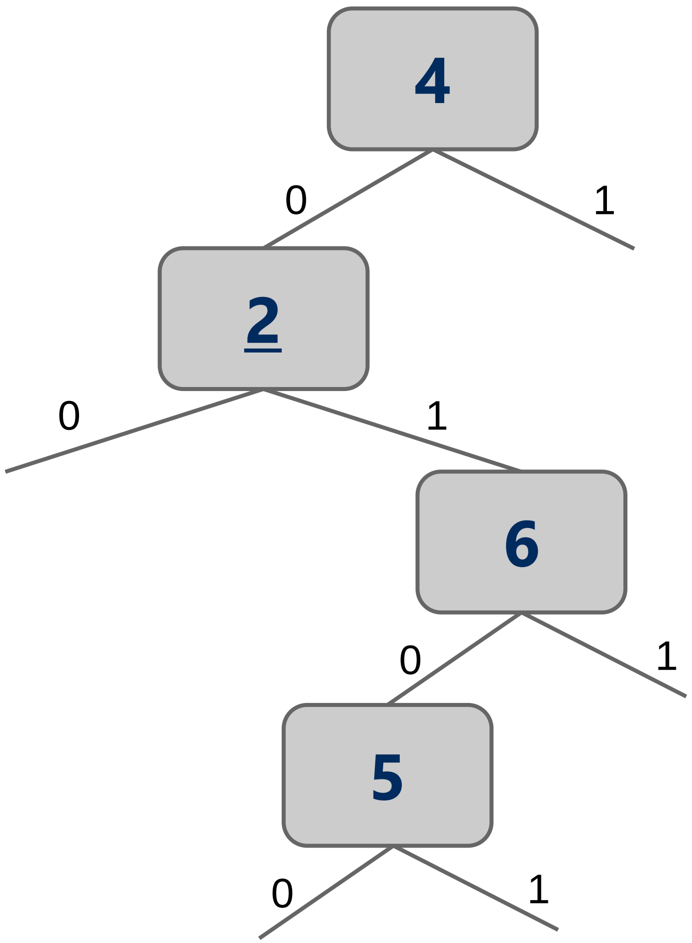
DST example: Delete

- Delete 3
- Can replace it with any leaf in its subtree
- Let's replace it with 2
- OK because 2 shares "0" as a prefix with 3, so it also shares "0" as a prefix with 6 and 5



DST example: Delete

- Delete 3
- Can replace it with any leaf in its subtree
- Let's replace it with 2
- OK because 2 shares "0" as a prefix with 3, so it also shares "0" as a prefix with 6 and 5



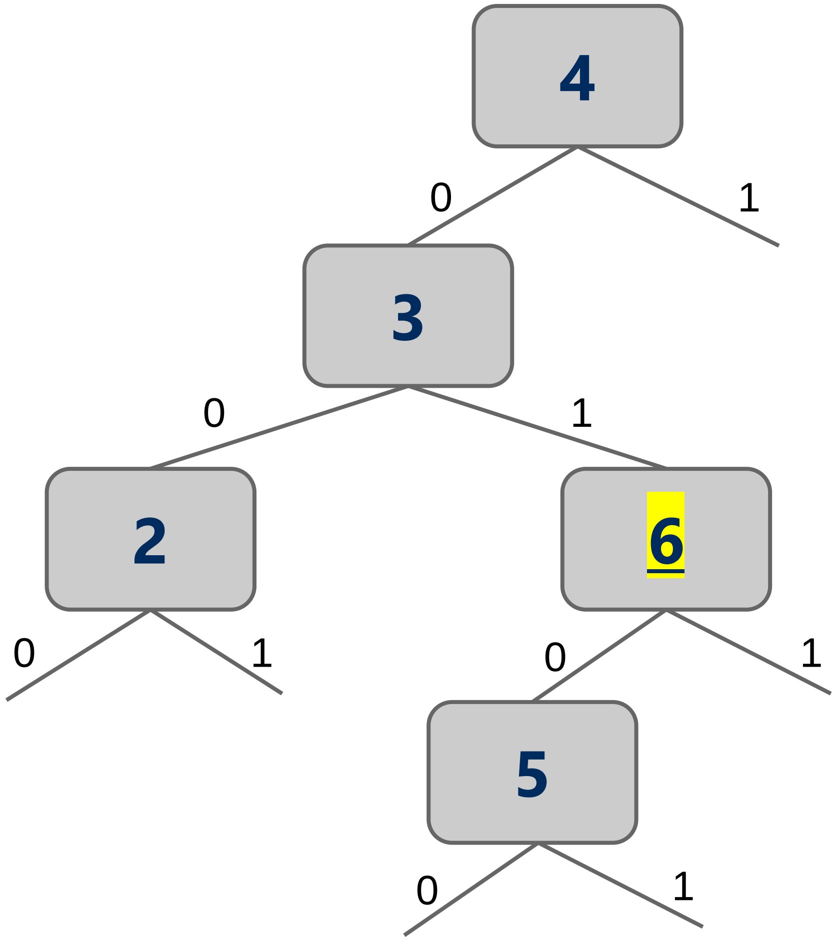
DST example: Variable length keys

- Insert

1 01

- Must be in place of 6

- Replace 6 by 1 and re-insert 6



DST example: Variable length keys

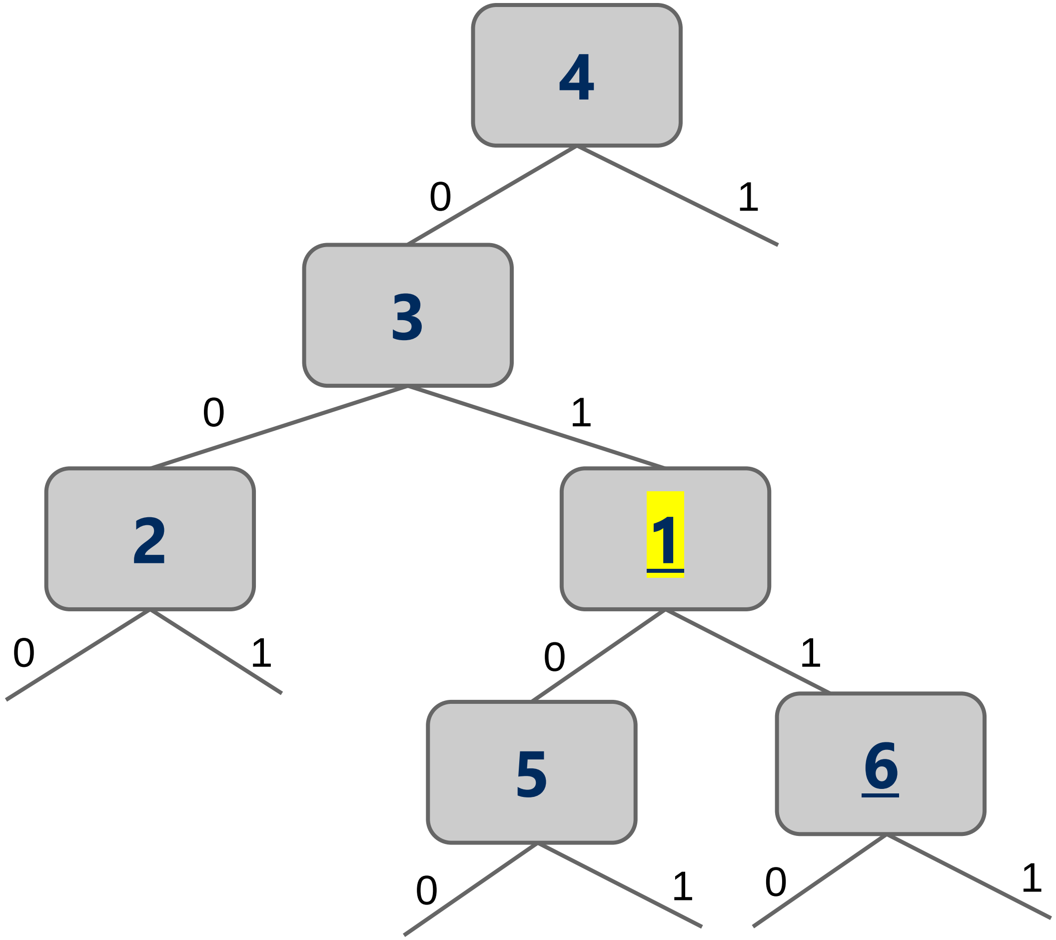
- Insert

1 01

- Must be in place of 6

- Replace 6 by 1 and re-insert

6 0110



Analysis of Digital Search Trees

- We end up doing many **equality** comparisons against the full key
- This is better than less than/greater than comparison in BST
- Can we improve on this?

Radix search tries (RSTs)

- Trie as in retrieve, pronounced the same as “try”
- Instead of storing keys inside nodes in the tree, we store them implicitly as paths down the tree
  - Interior nodes of the tree only serve to direct us according to the bitstring of the key
  - Values can then be stored at the end of key’s bitstring path (i.e., at leaves)
  - RST uses less space than BST and DST

Adding to Radix Search Trie (RST)

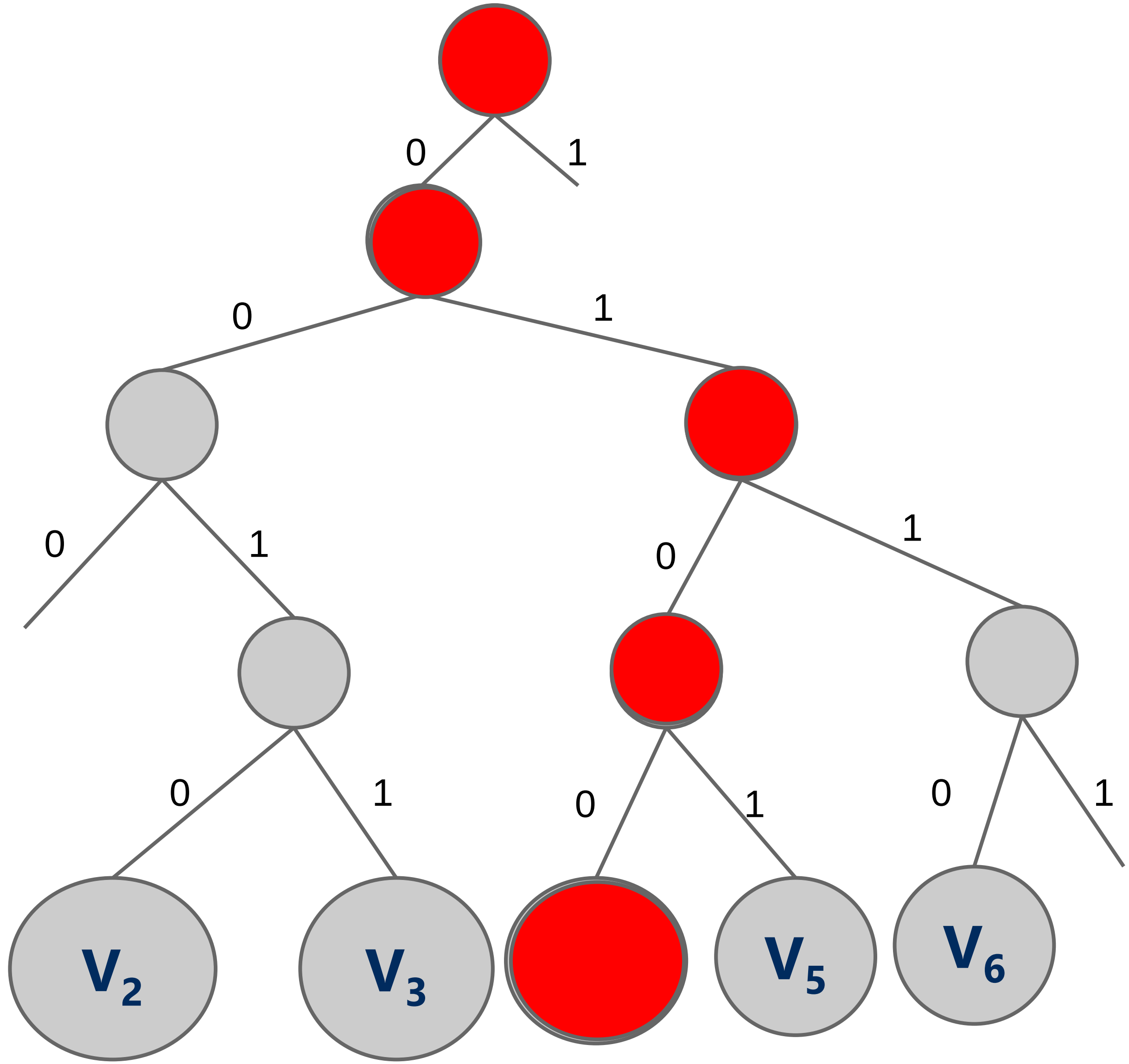
- Input: key and corresponding value
- if root is null, set root ← new node
- current node ← root
- for each *bit* in the key
  - if bit == 0,
    - if current.left is null, set current.left = new node
    - move to left child
      - set current ← current.left
  - if bit == 1,
    - if current.right is null, set current.right = new node

    current ← current.right
- current.value = value

RST example

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101



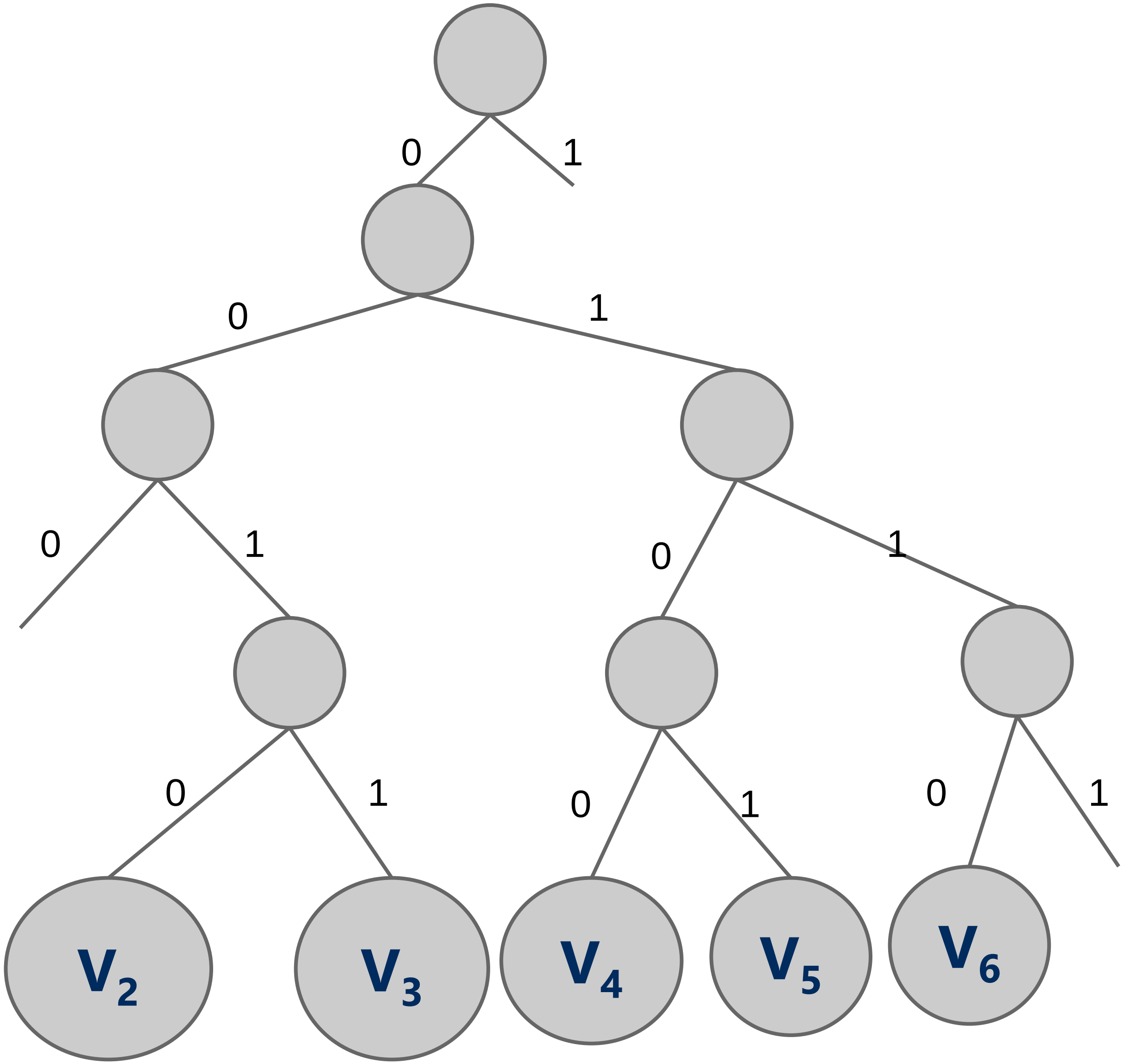
Searching in Radix Search Trie (RST)

- Input: key
- current node ← root
- for each *bit* in the key
  - if current node is null, return *key not found*
  - if bit == 0,
    - current ← current.left
  - if bit == 1,
    - current ← current.right
- if current node is null or the value inside is null
  - return key *not found*
- else return current.value

RST example

Search:

3 0011

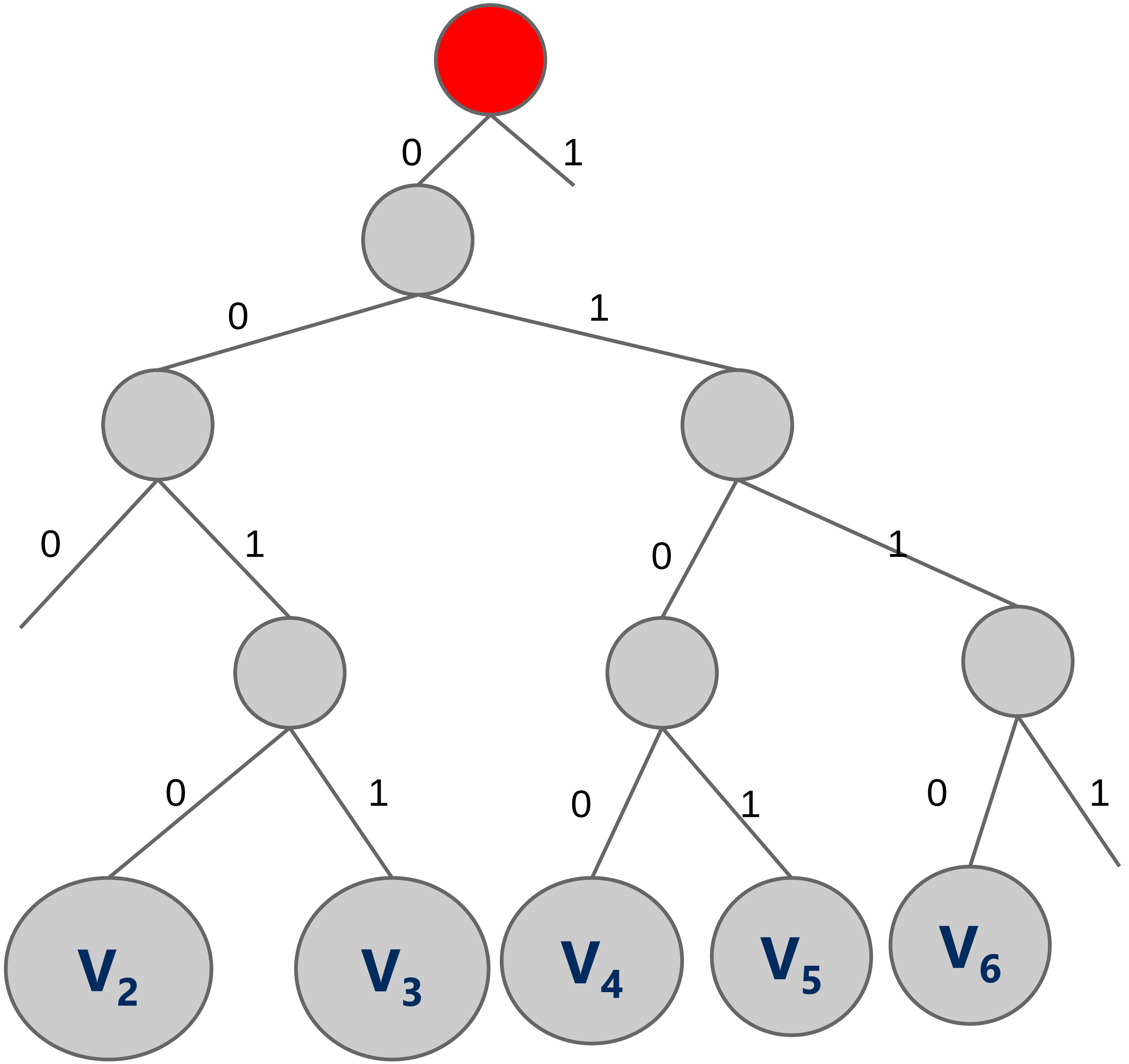


RST example

Search:

3 0011

7 0111

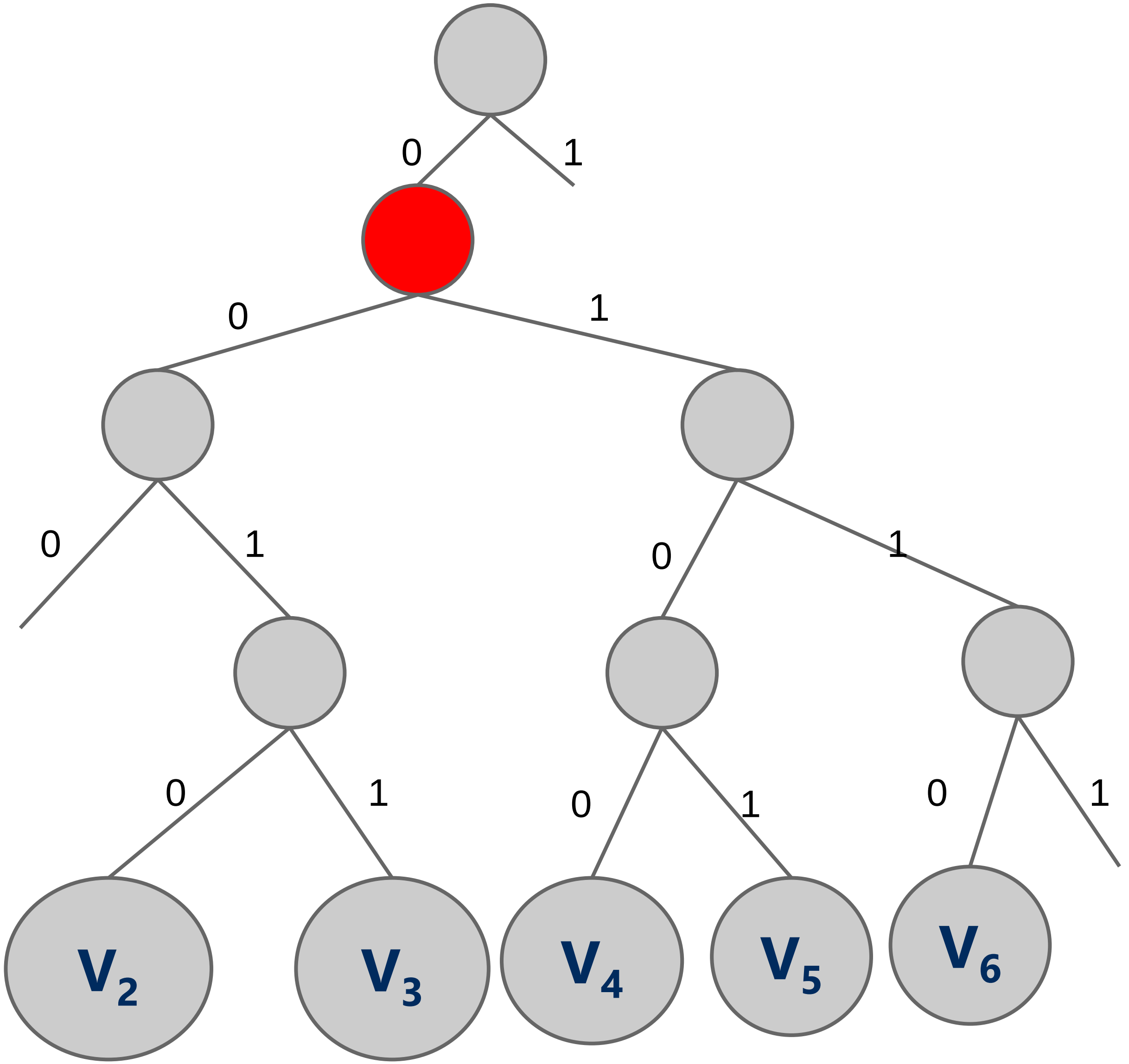


RST example

Search:

3 0011

7 0111

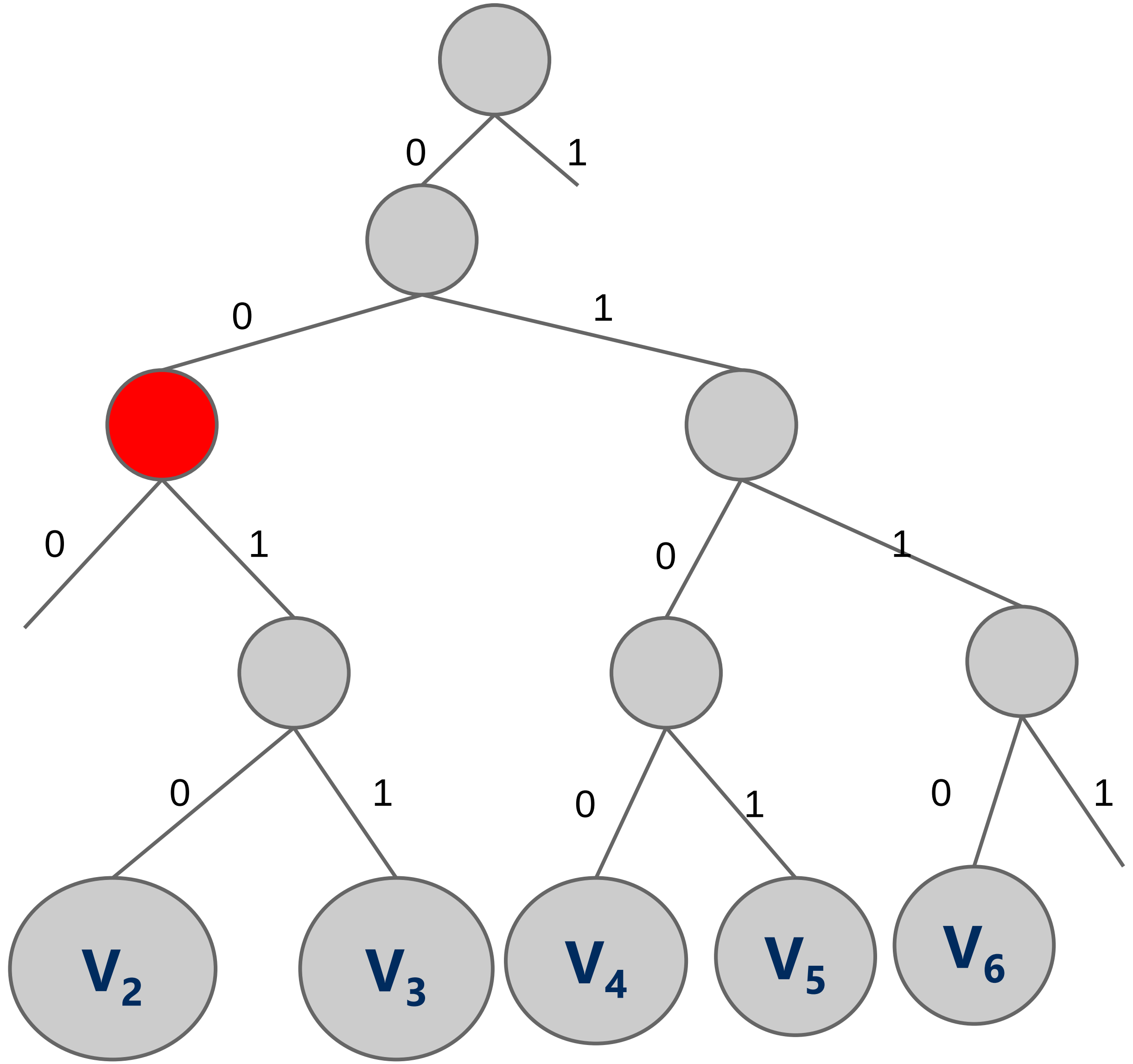


RST example

Search:

3 0011

7 0111

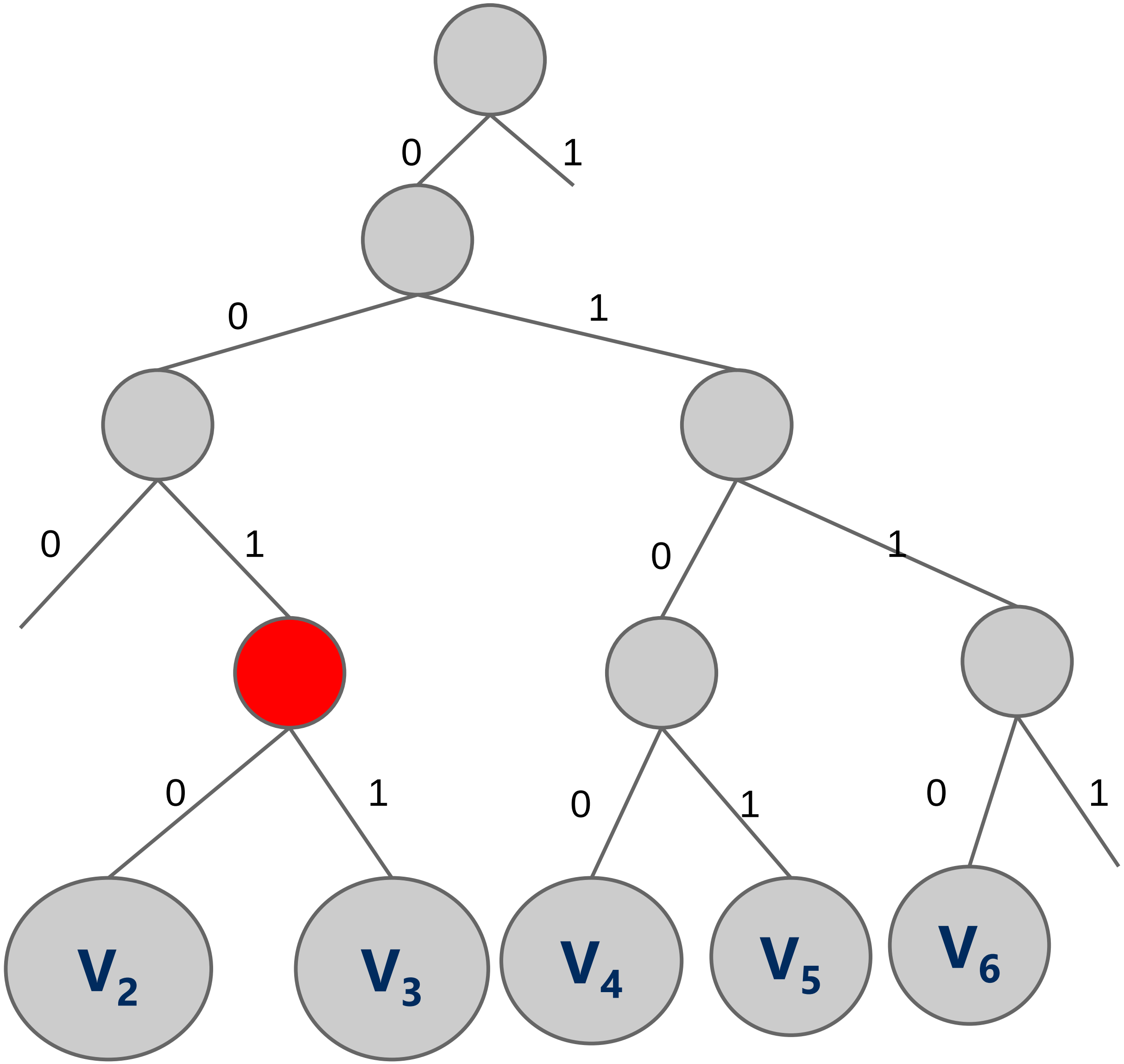


RST example

Search:

3 0011

7 0111

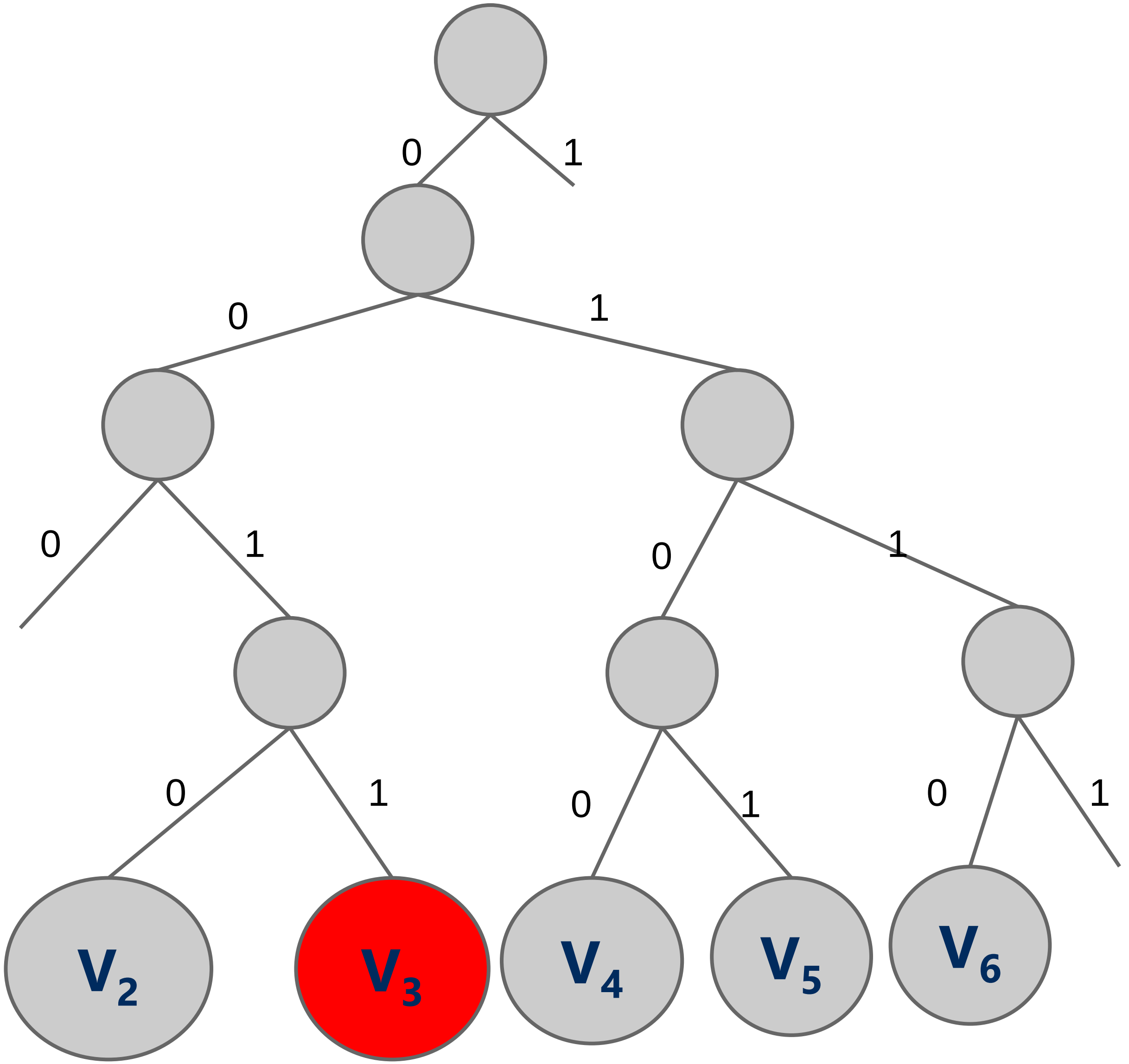


RST example

Search:

3 0011

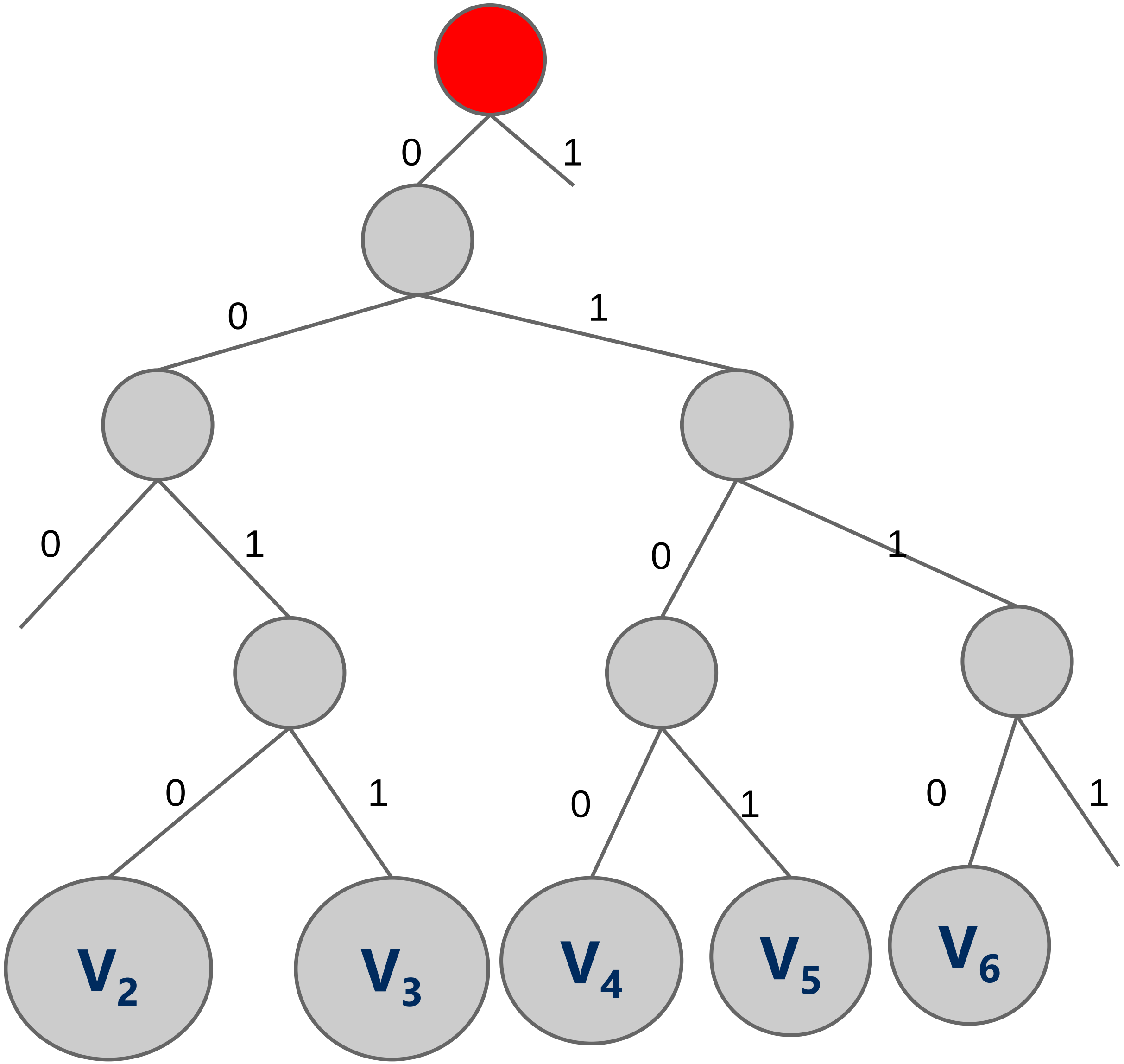
7 0111



RST example

Search:

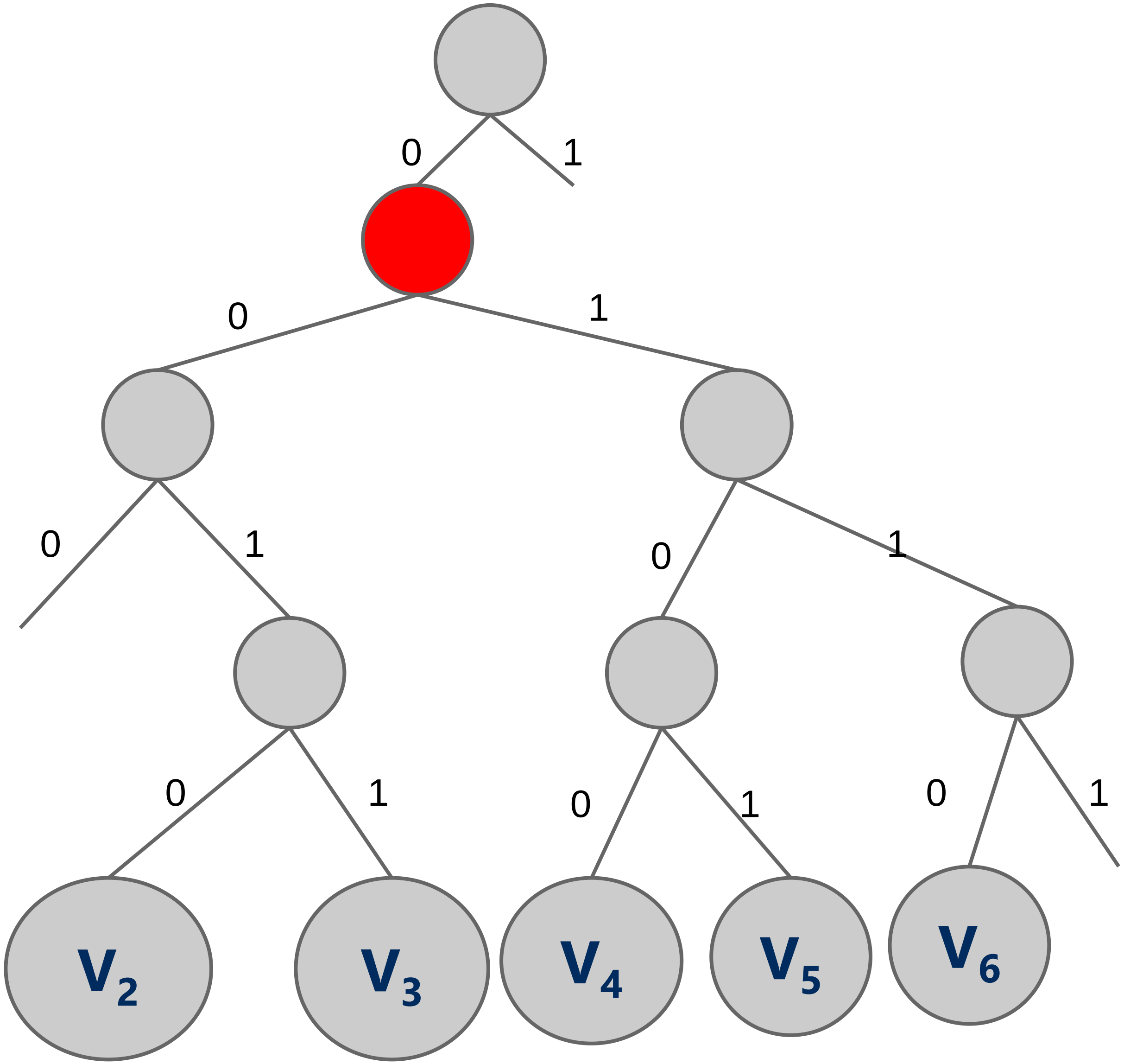
7 0111



RST example

Search:

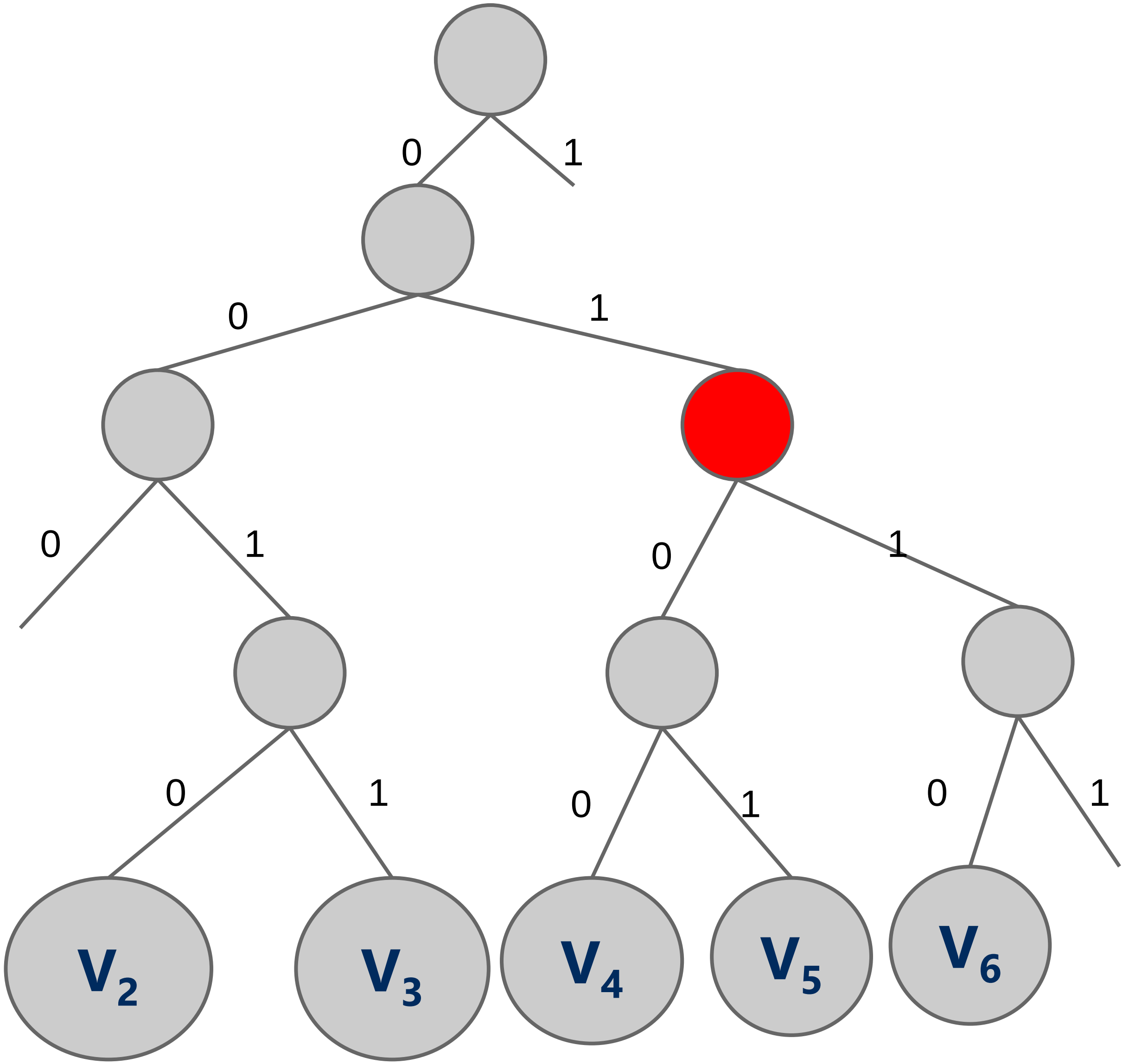
7 0111



RST example

Search:

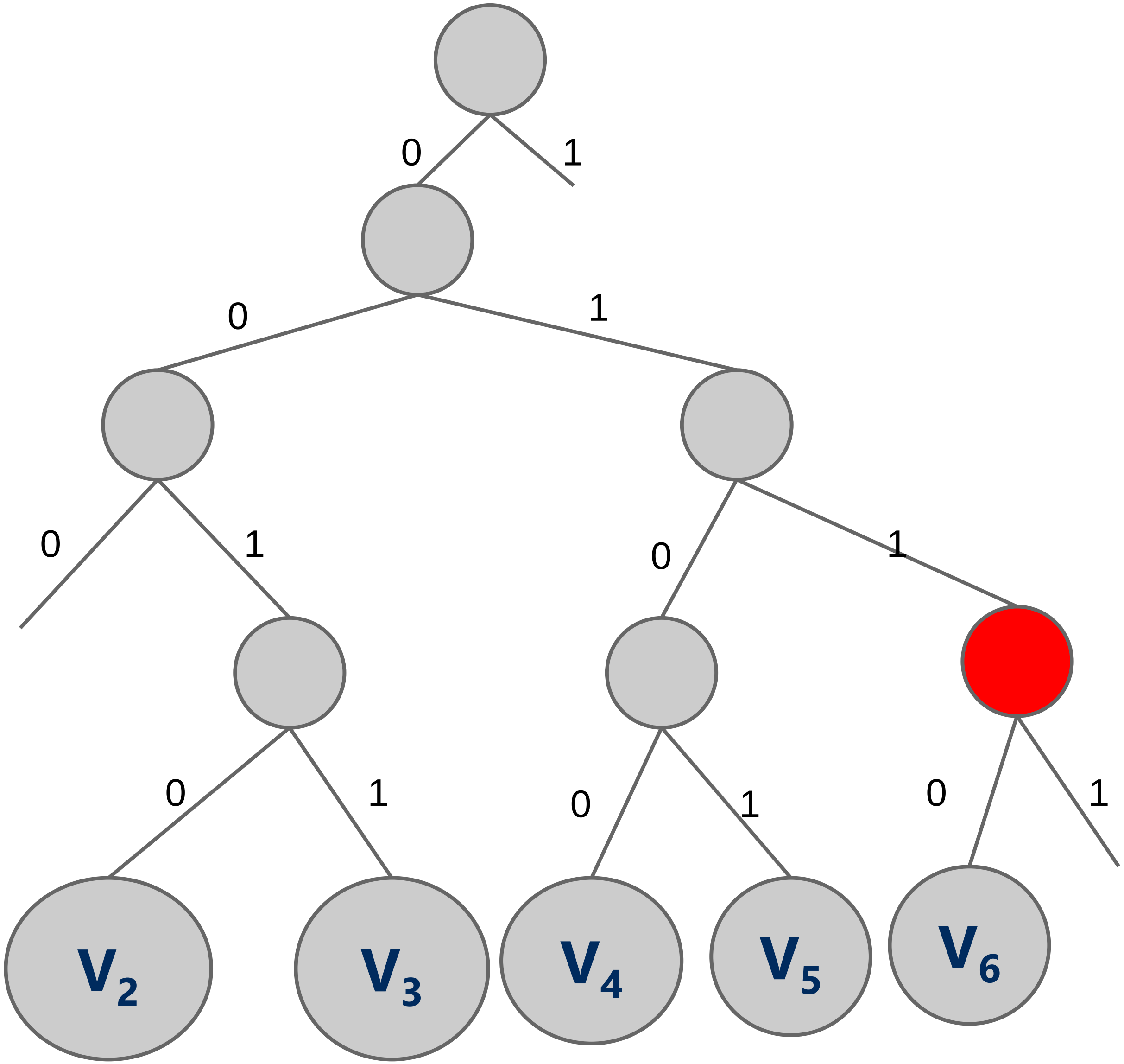
7 0111



RST example

Search:

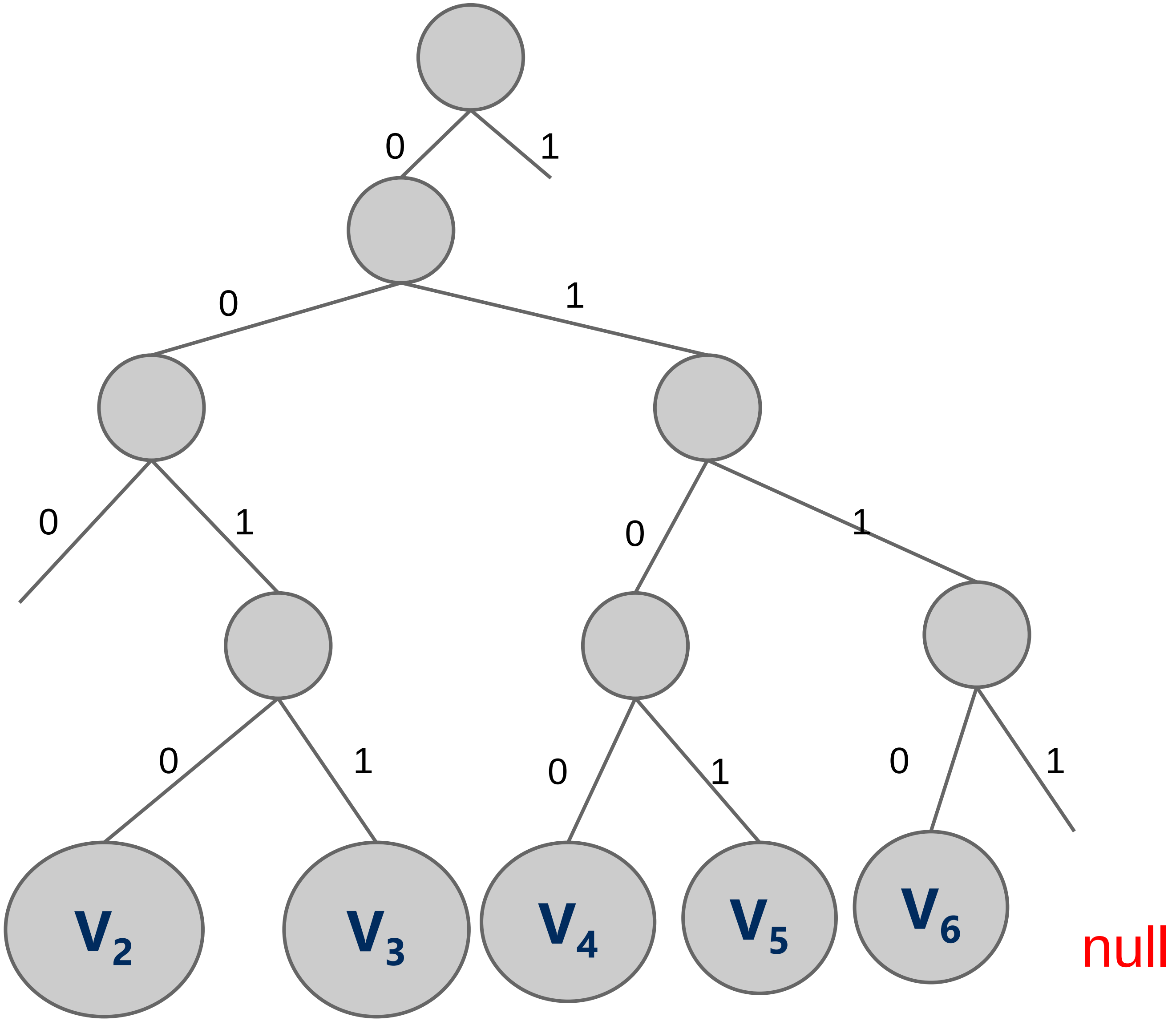
7 0111



RST example

Search:

7 0111



RST Runtime analysis

- **for add:**
  - Theta(b): b is the bit length of the key
  - However, this time we don't have full key comparisons
- for search:
  - **search hit**
    - Theta(b)
  - **search miss**
    - maybe less than Theta(b)

RST Limitations

- Would this structure work as well for other key data types?
- Characters?
    - Characters are the same as 8-bit ints (assuming simple ascii)
- Strings?
- May have huge bit lengths
- How to store Strings?

Larger branching factor tries

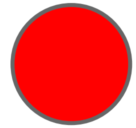
- In our binary-based Radix search trie, we considered one bit at a time
- What if we applied the same method to characters instead of bits in a string?
  - What would this new structure look like?
  - How many children per node?
    - up to R (the alphabet size)
  - Also called R-way radix search tries

Adding to R-way Radix RST

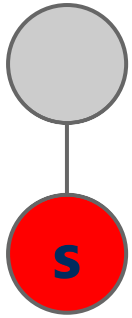
- if root is null, set root ← new node
- current node ← root
- for each *character c* in the key
  - Find the c*th* child of current
    - if child is null, create a new node and attach as the c*th* child
    - move to child
      - current ← child
- insert value into current node

Another trie example

she

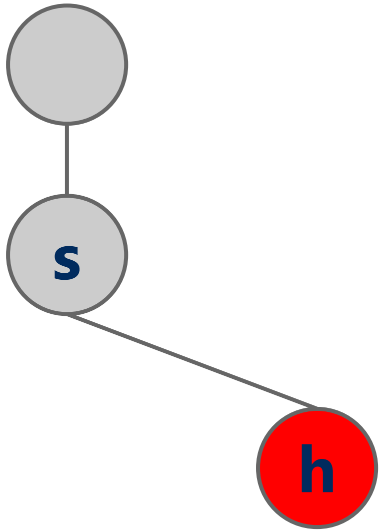


Another trie example



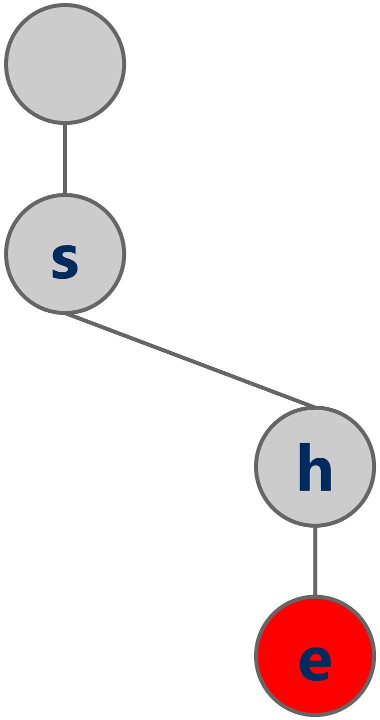
she

Another trie example



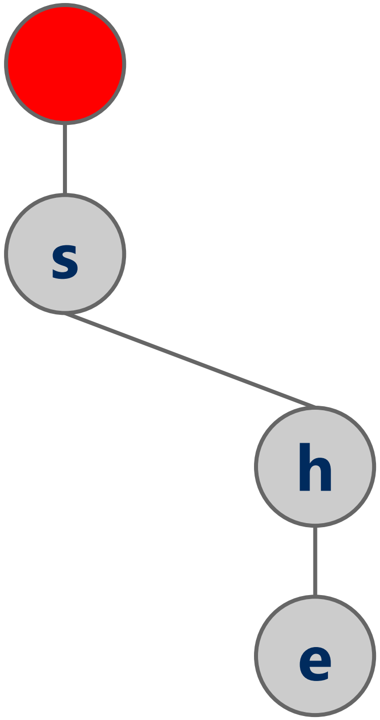
she

Another trie example



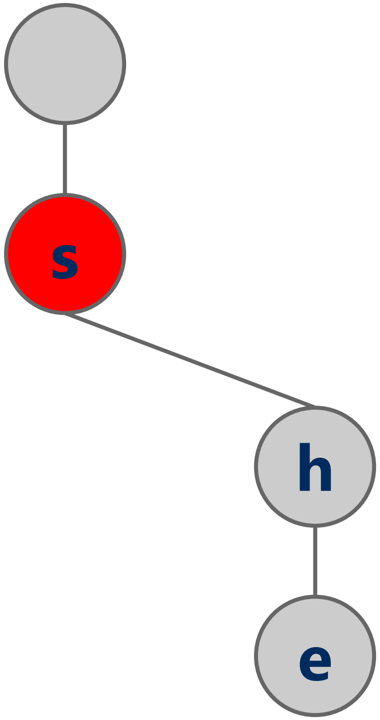
she

Another trie example



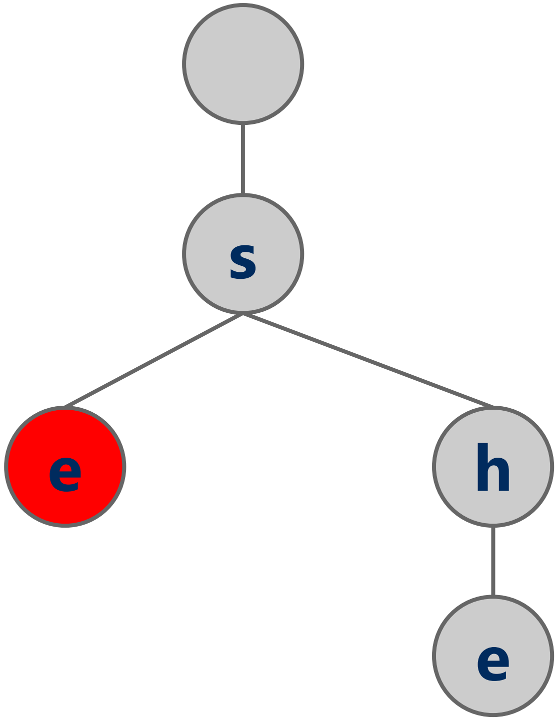
sells

Another trie example



sells

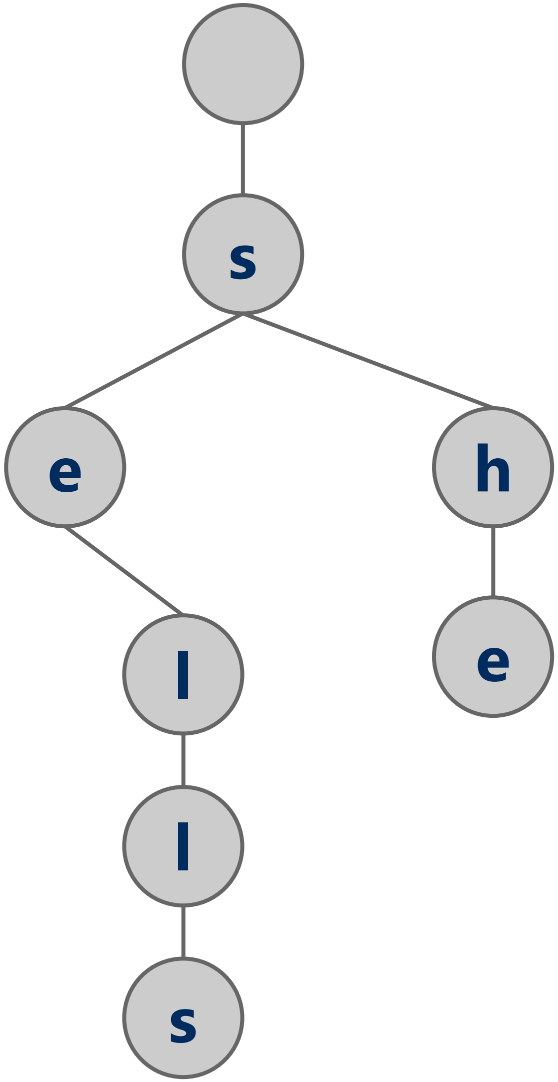
Another trie example



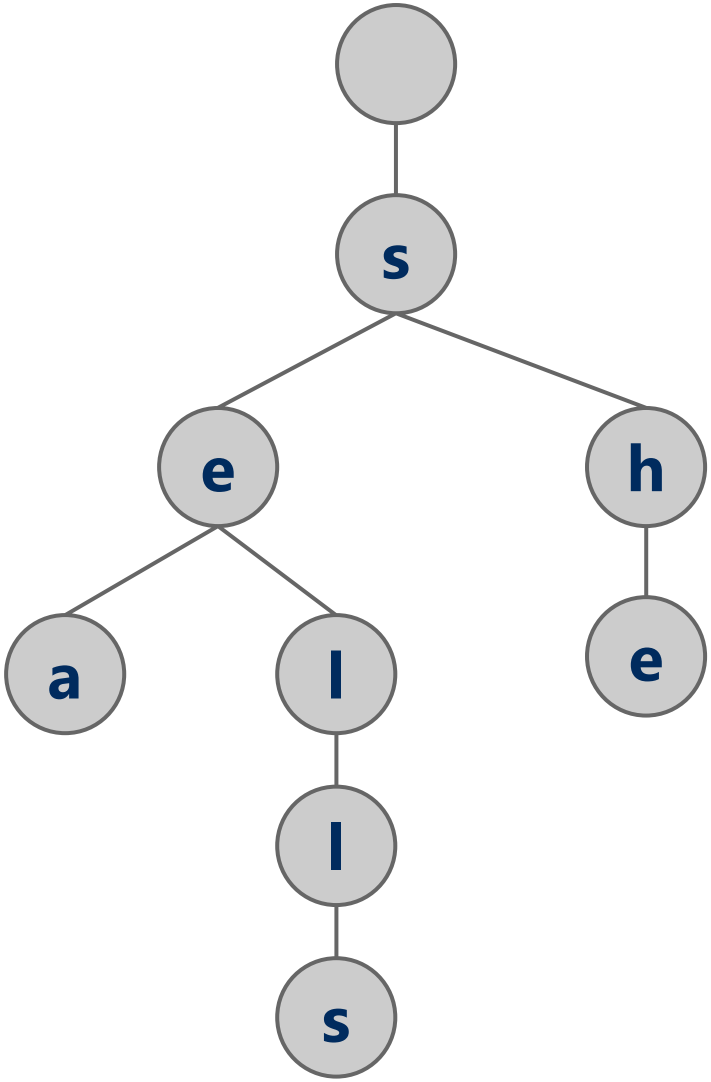
sells

Another trie example

sells



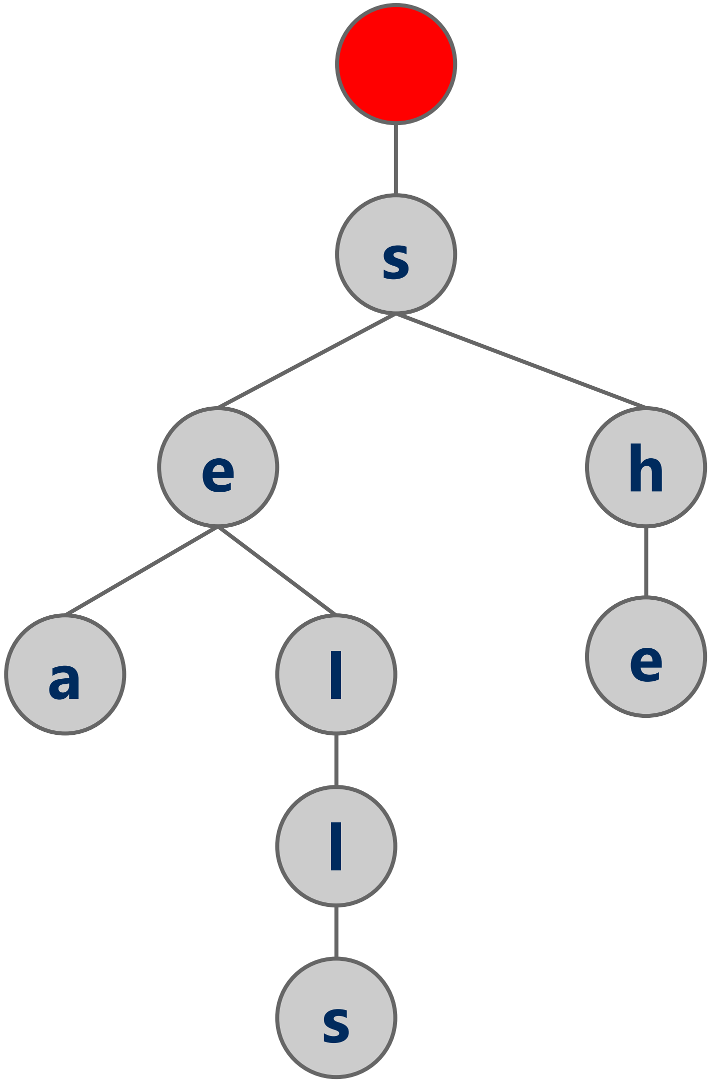
Another trie example



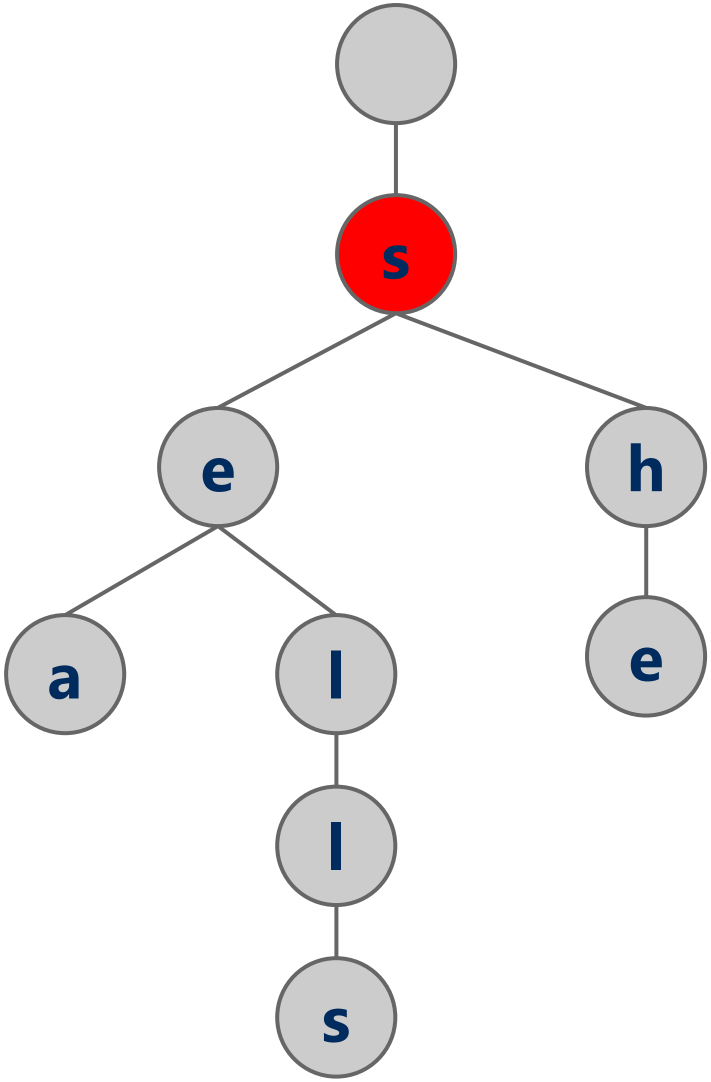
sea

Another trie example

shells

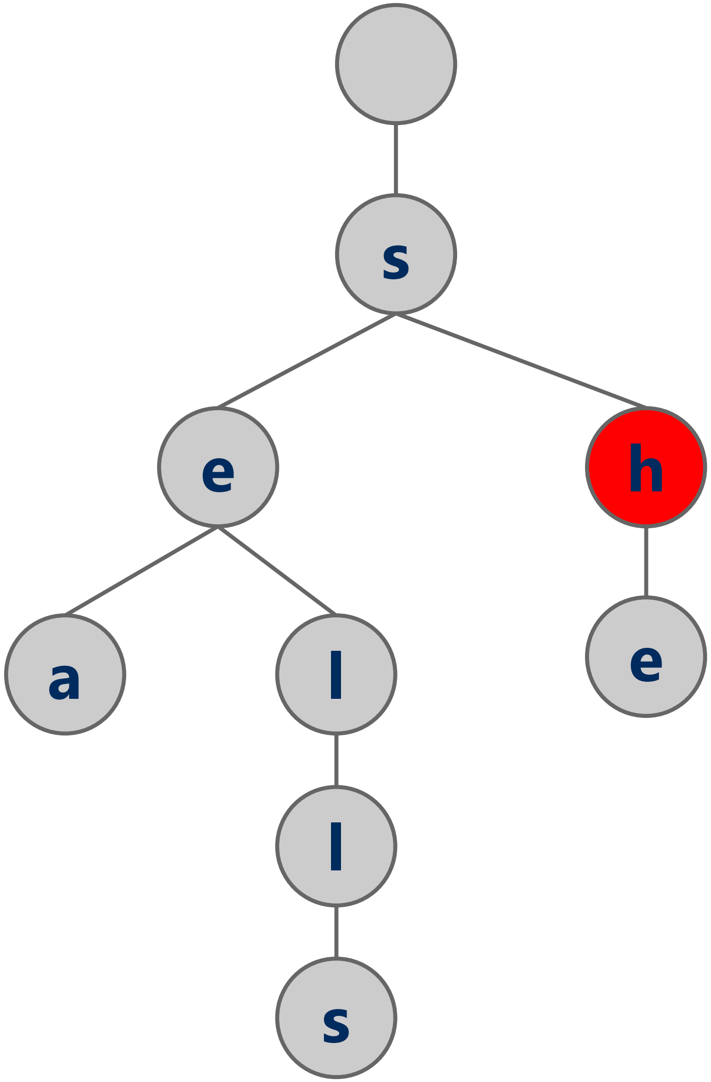


Another trie example



shells

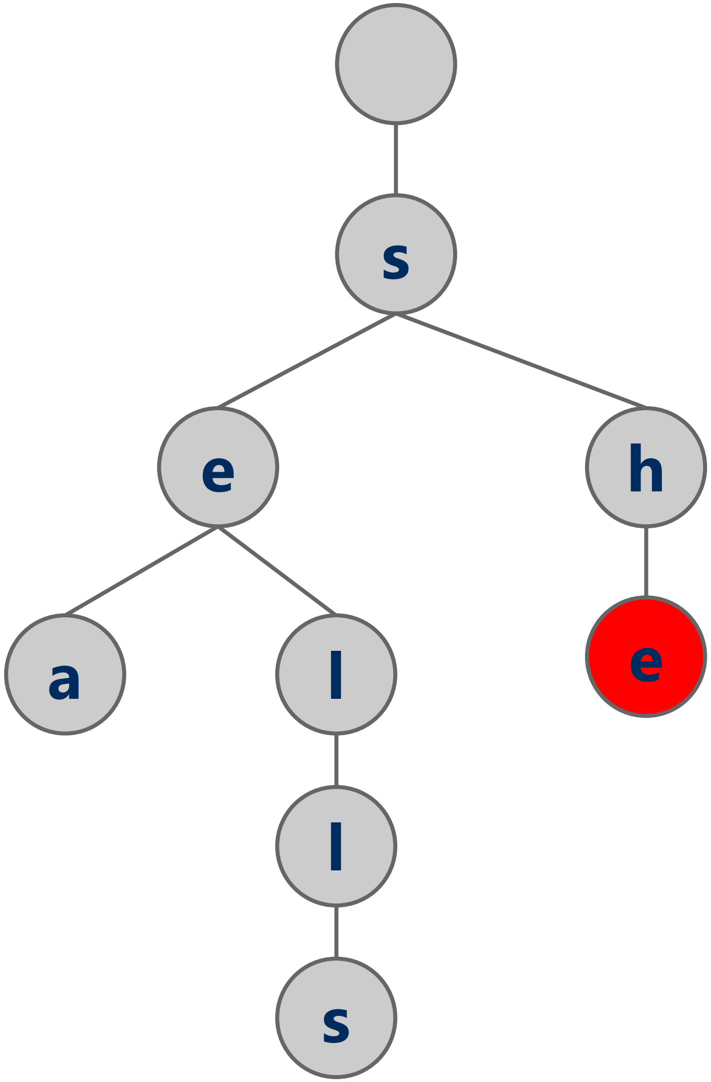
Another trie example



shells

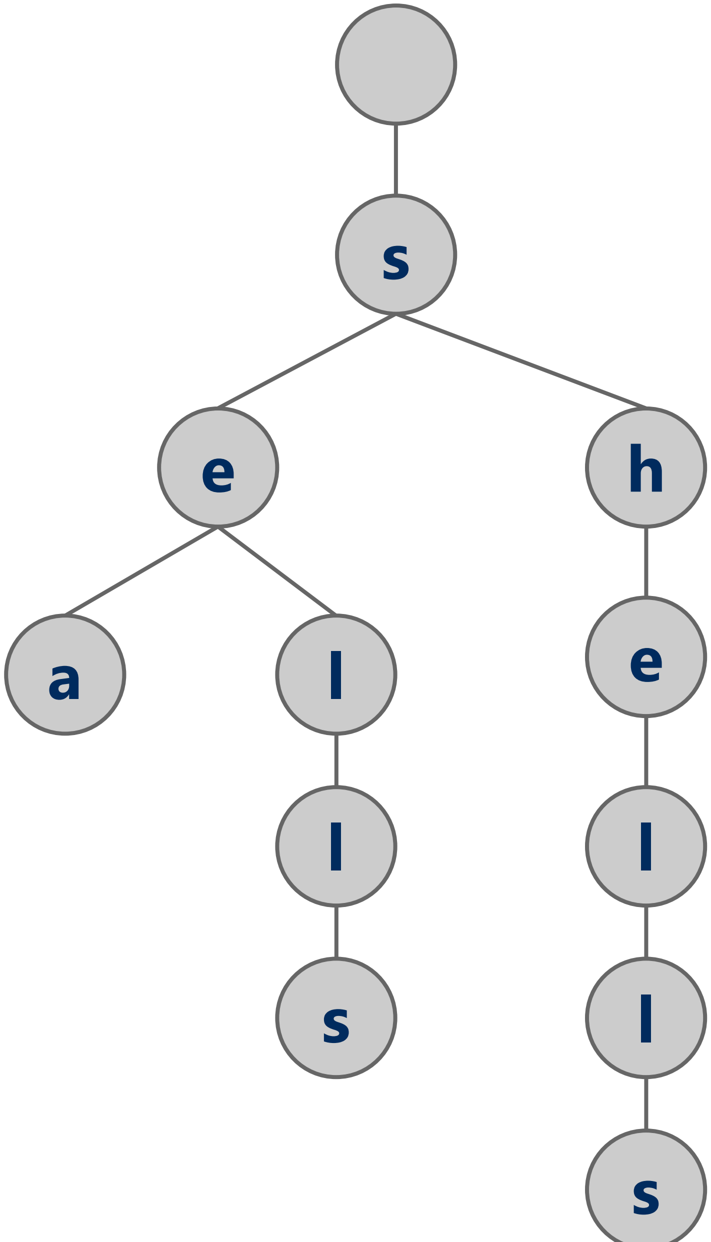
Another trie example

shells

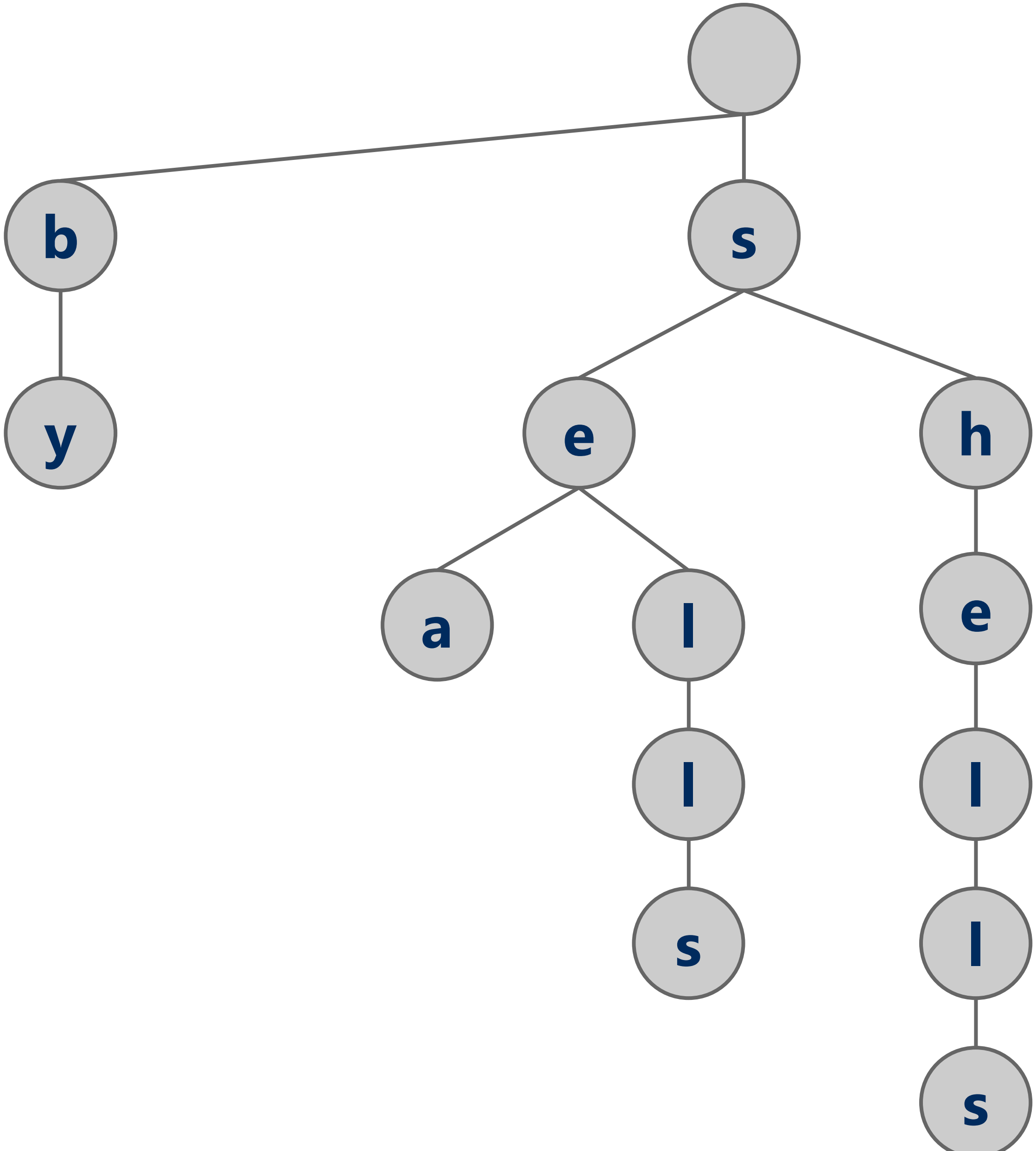


Another trie example

shells

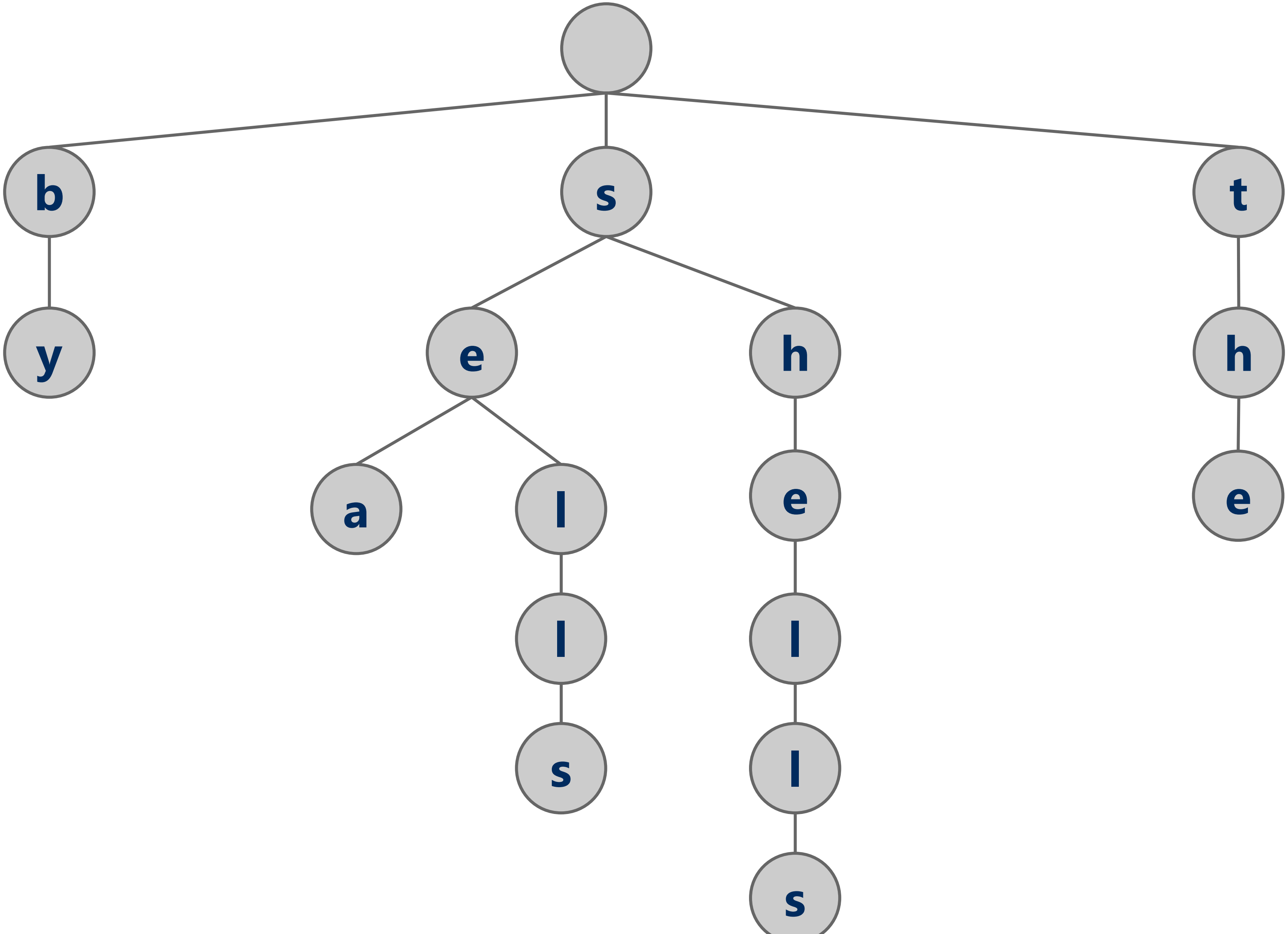


Another trie example



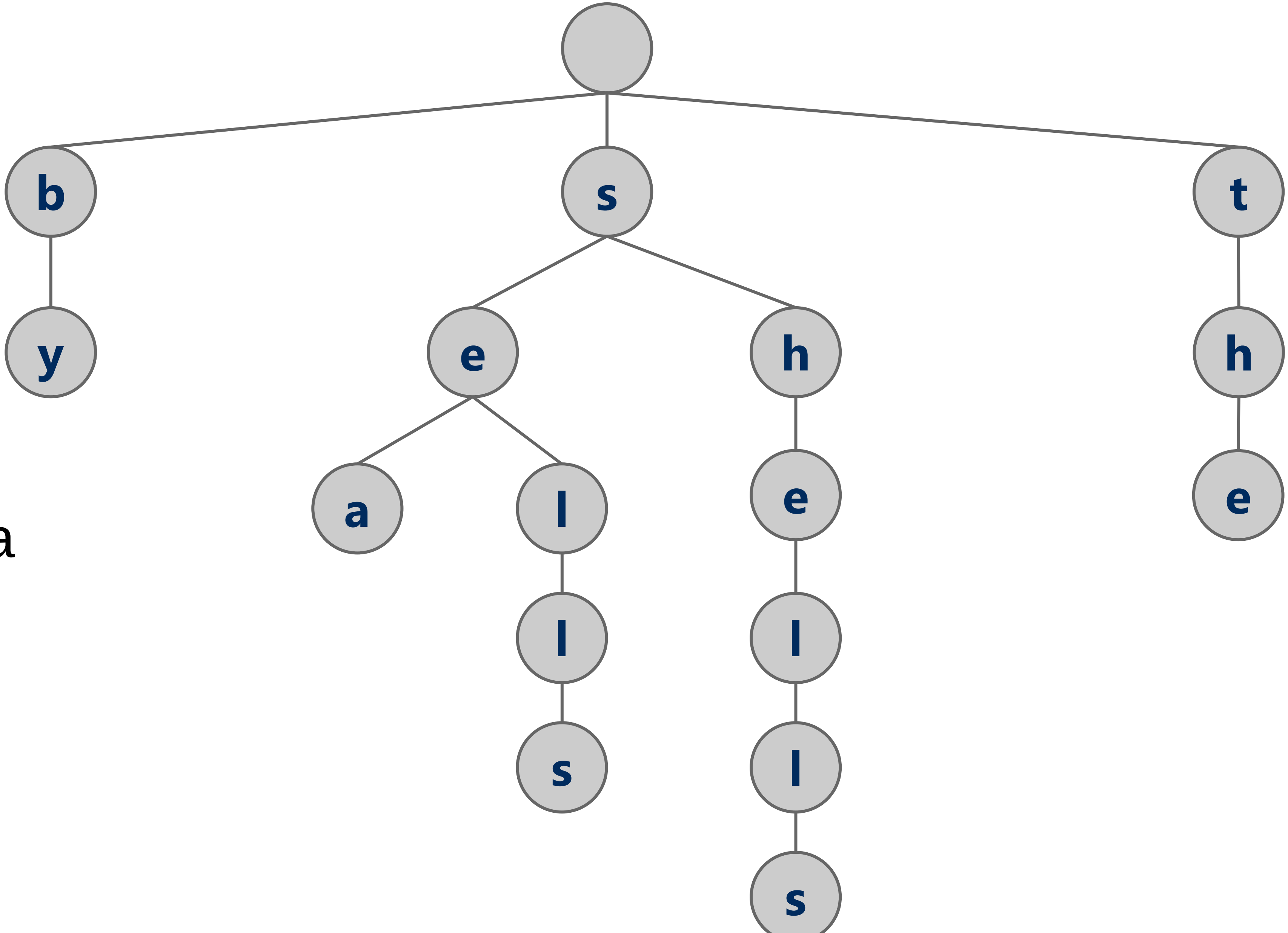
by

Another trie example



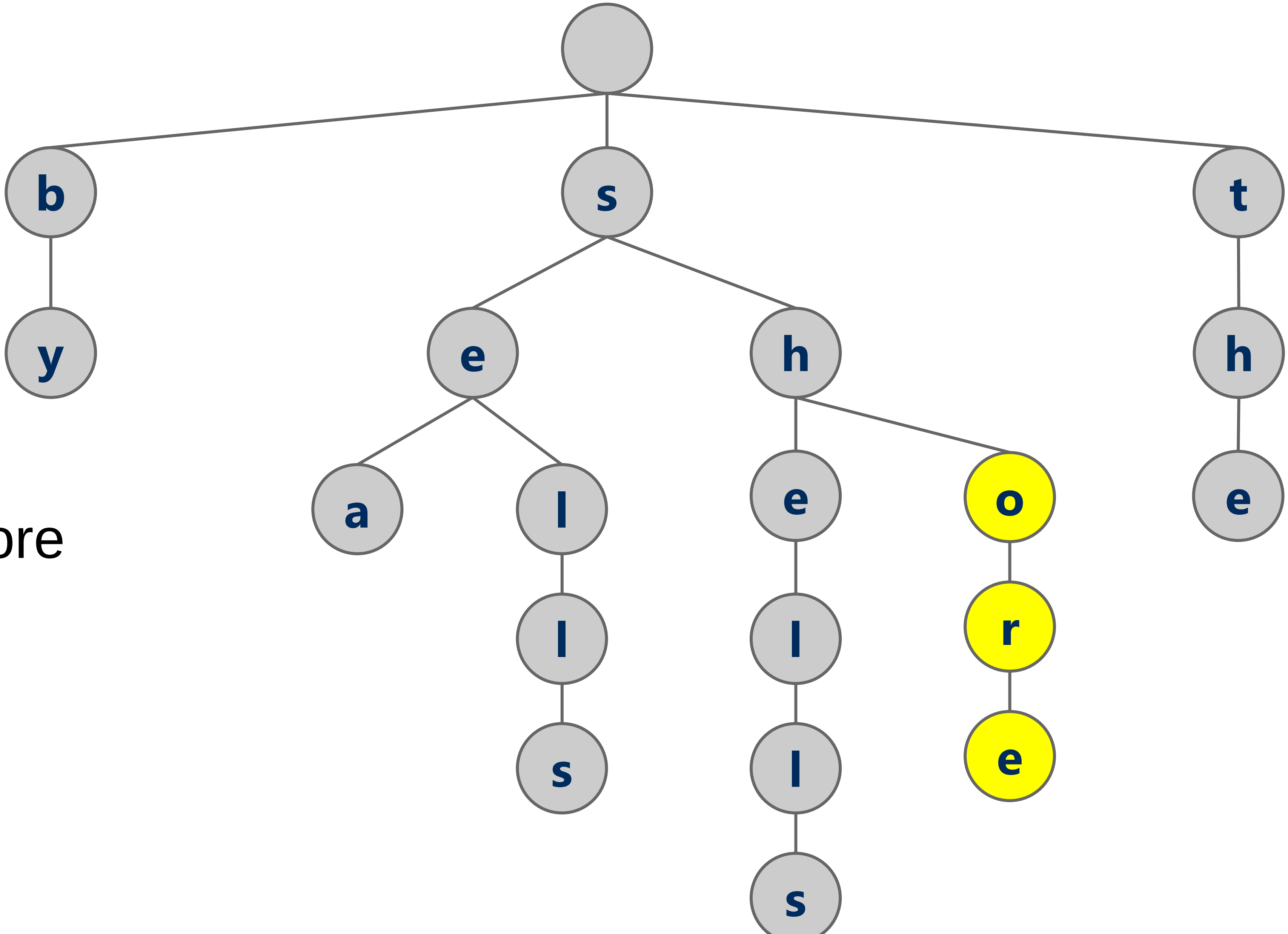
the

Another trie example



sea

Another trie example



shore

Analysis

- Runtime of add and *search hit*?
- O(w) where w is the character length of the string
  - So, what do we gain over RSTs?
    - $w < b$
    - e.g., assuming fixed-size encoding $w = \frac{b}{\lceil \log R \rceil}$
    - tree height is reduced

Search Miss

- Search Miss time for R-way RST
  - Require an average of $log_R(n)$ nodes to be examined
    - Proof in Proposition H of Section 5.2 of the text
- Average tree height with 2^{20} keys in an RST?
  - $\log_2 n = \log_2 2^{20} = 20$
- With 2^{20} keys in a large branching factor trie, assuming 8-bits at a time?
  - $\log_R n = \log_{256} 2^{20} = \log_{256} (2^8)^{2.5} = = \log_{256} 256^{2.5} = 2.5$

Implementation Concerns

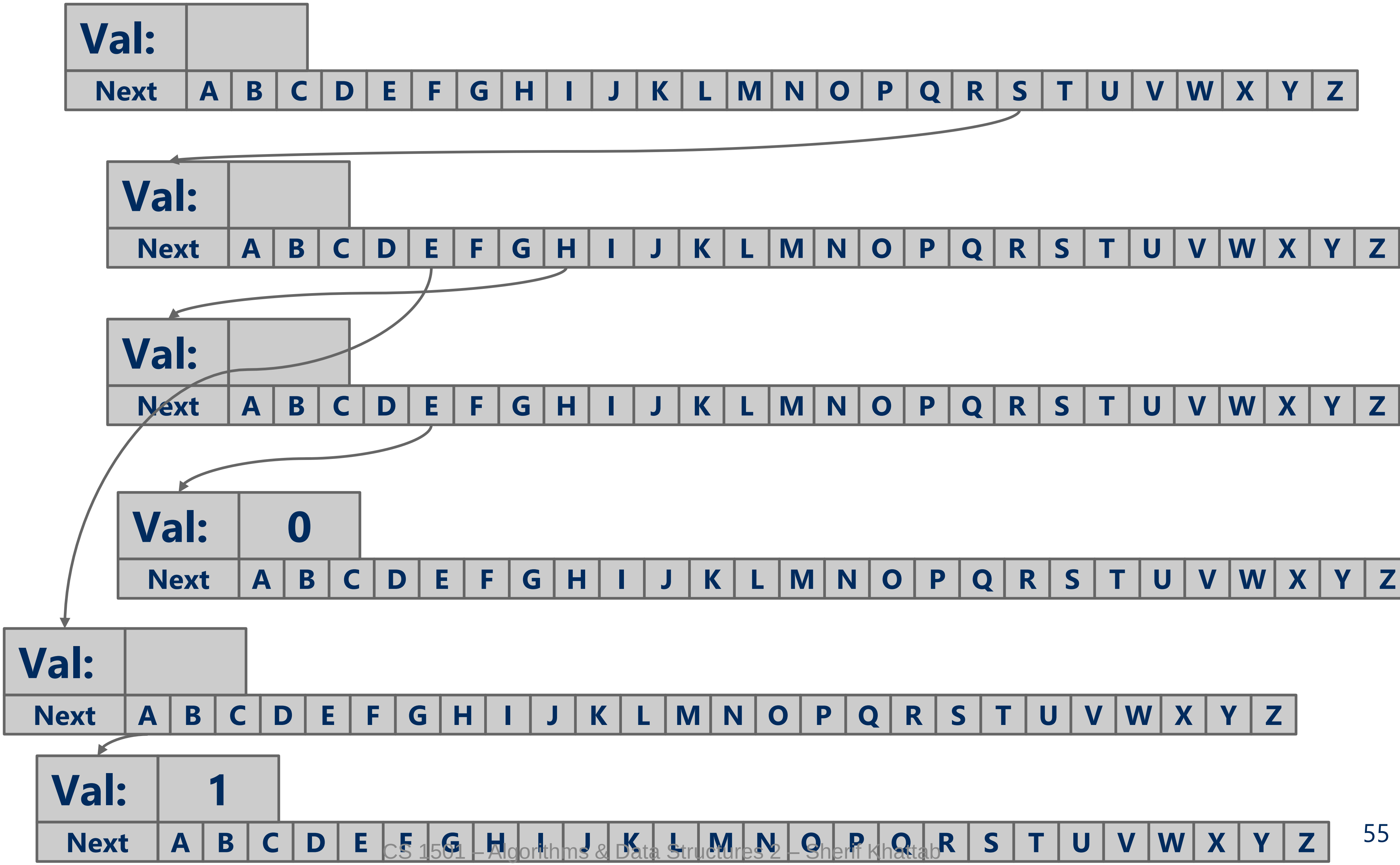
- See TrieSt.java
  - Implements an R-way trie
- Basic node object:

Where R is the branching factor

```
private class Node {
    private Object val;
    private Node[] next;
    private Node(){
      next = new Node[R];
    }
}
```

- Non-null **val** means we have traversed to a valid key
- Again, note that keys are not directly stored in the trie at all

R-way trie example

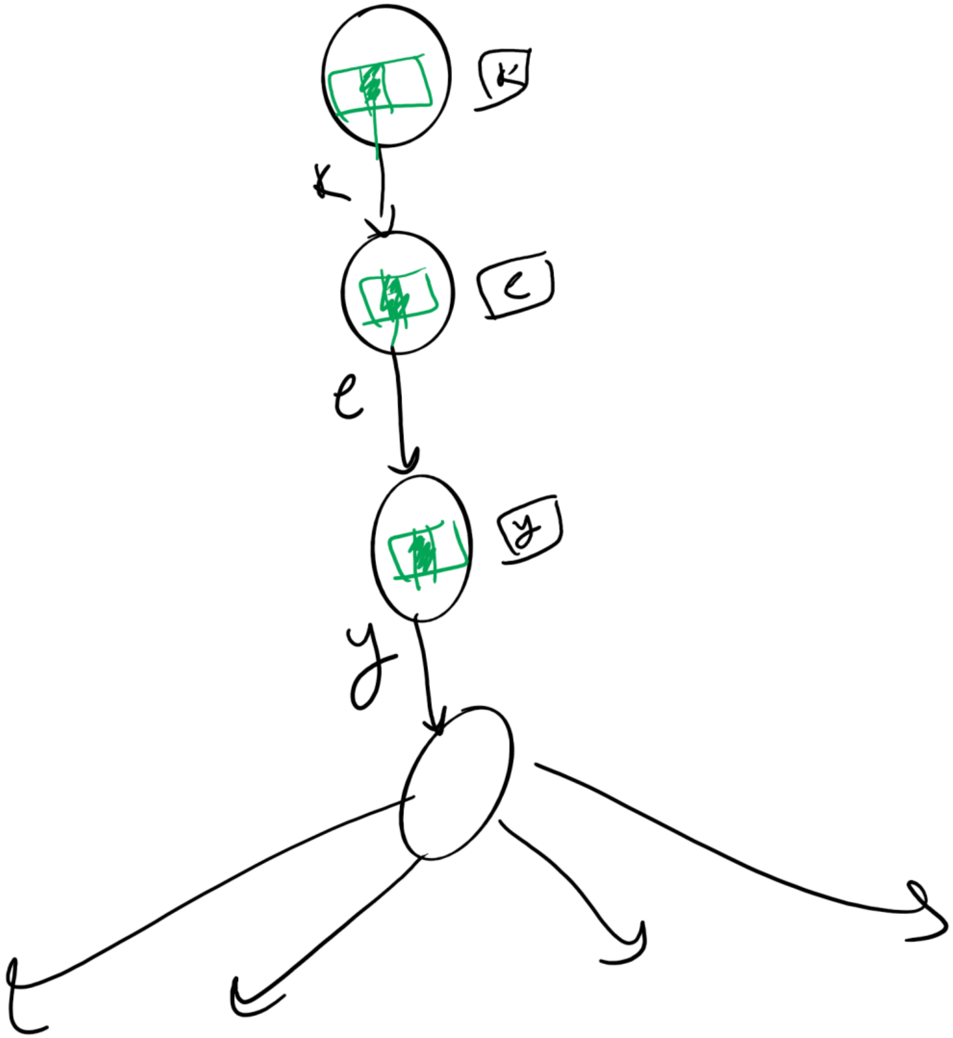


Summary of running time

	insert	search hit	search miss
binary RST	$\Theta(b)$	$\Theta(b)$	$\Theta(\log_2 n)$ on average
multi-way RST	$\Theta(w)$	$\Theta(w)$	$\Theta(\log_R n)$

R-way RST's nodes may waste space!

- Considering 8-bit ASCII, each node contains 2^8 references!
- This is especially problematic as in many cases, a lot of this space is wasted
  - Common paths or prefixes for example, e.g., if all keys begin with "key", thats 255*3 wasted references!
  - At the lower levels of the trie, most keys have probably been separated out and reference lists will be sparse



Solution: De La Briandais tries (DLBs)

Main idea: replace the array inside the node of the R-way trie with a linked-list

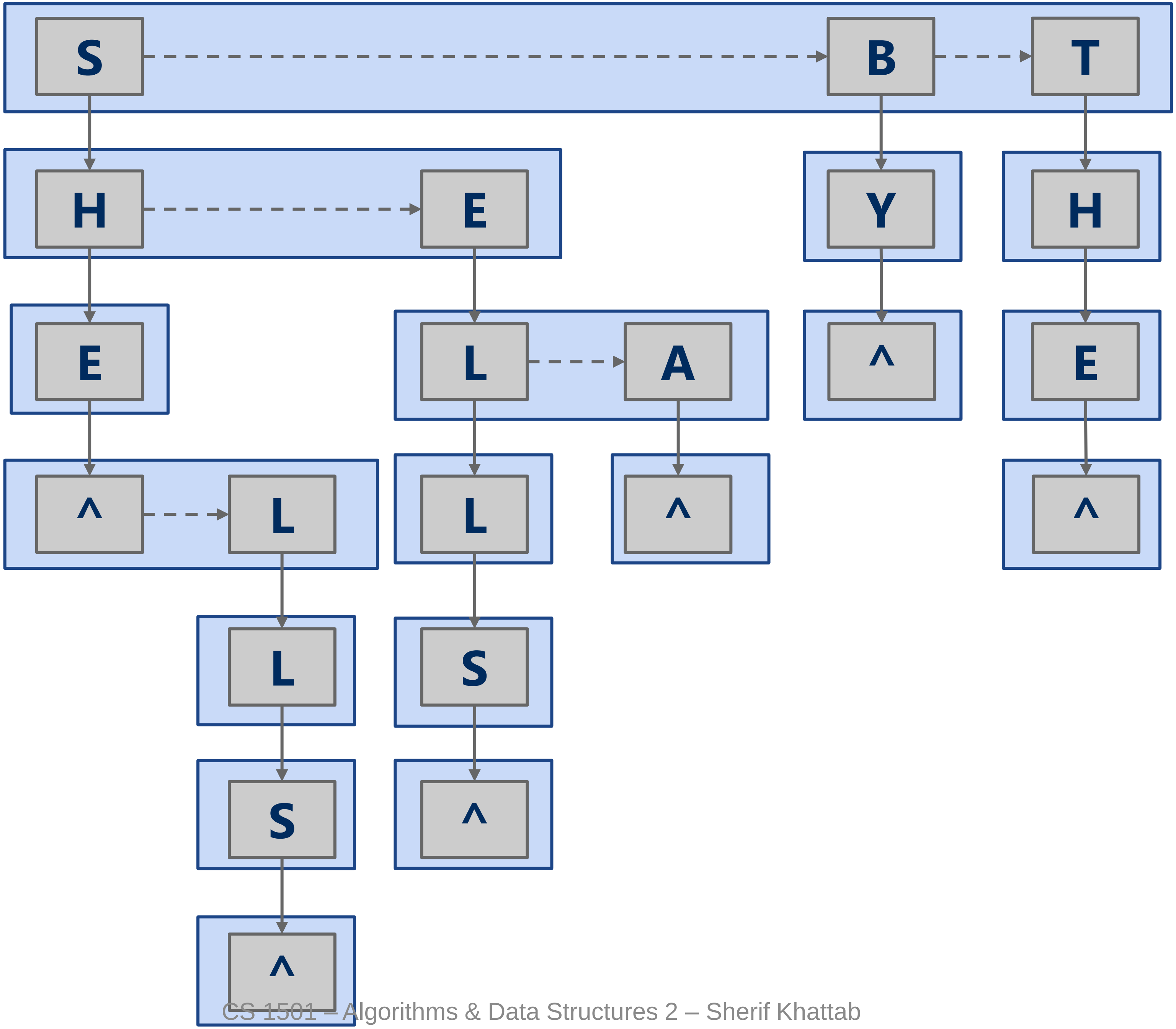
De La Briandais (DLB) Trie

- tree-like structure used for searching when keys are sequences of characters
- each nodelet
  - stores one character,
  - points to a sibling (linked list of siblings), and
  - points to a child

Adding to DLB Trie

- if root is null, set root ← new node
- current node ← root
- for each *character c* in the key
  - Search for *c* in the linked list headed at current using sibling links
    - if not found, create a new node and attach as a sibling to the linked list
  - move to child of the found node
    - either recursively or by current ← child
- if at last character of key, insert value into current node and return

DLB Example



DLB analysis

- How does DLB performance differ from R-way tries?
- Which should you use?

	search hit insert	
R-way RST	$\theta(w)$	
DLB	$\theta(wR)$	

De La Briandais (DLB) Trie

- worst-case running time is O(wR)
  - w: number of characters in the key
  - R: alphabet size
- worst-case can be avoided by using DLB only when the sibling lists are short

DLB vs. R-way RST: Space comparison example

- **Q: How does DLB save space over R-way RST?** Assume the following set of keys:
  - ksm1 … ksm9
- How big does an 256-way RST take vs. a DLB trie?

R-way RST

```
private class Node {
    private Object val;
    private Node[] next;
     private Node(){
        next = new Node[R];
     }
}
```

Each node takes 4*(R+1) = 4*257 = 1028 bytes, assuming 4 bytes per reference variable

R-way RST

Val:
null
Next
k
Val:
null
Next
s
Val:
null
Next
m
Val:
null
Next
1
2
3
4
5
6
7
8
9
Val:
V_{ksm1}
Next
Val:
V_{ksm2}
Next

R-way RST

We will end up with 4 + 9 = 13 nodes

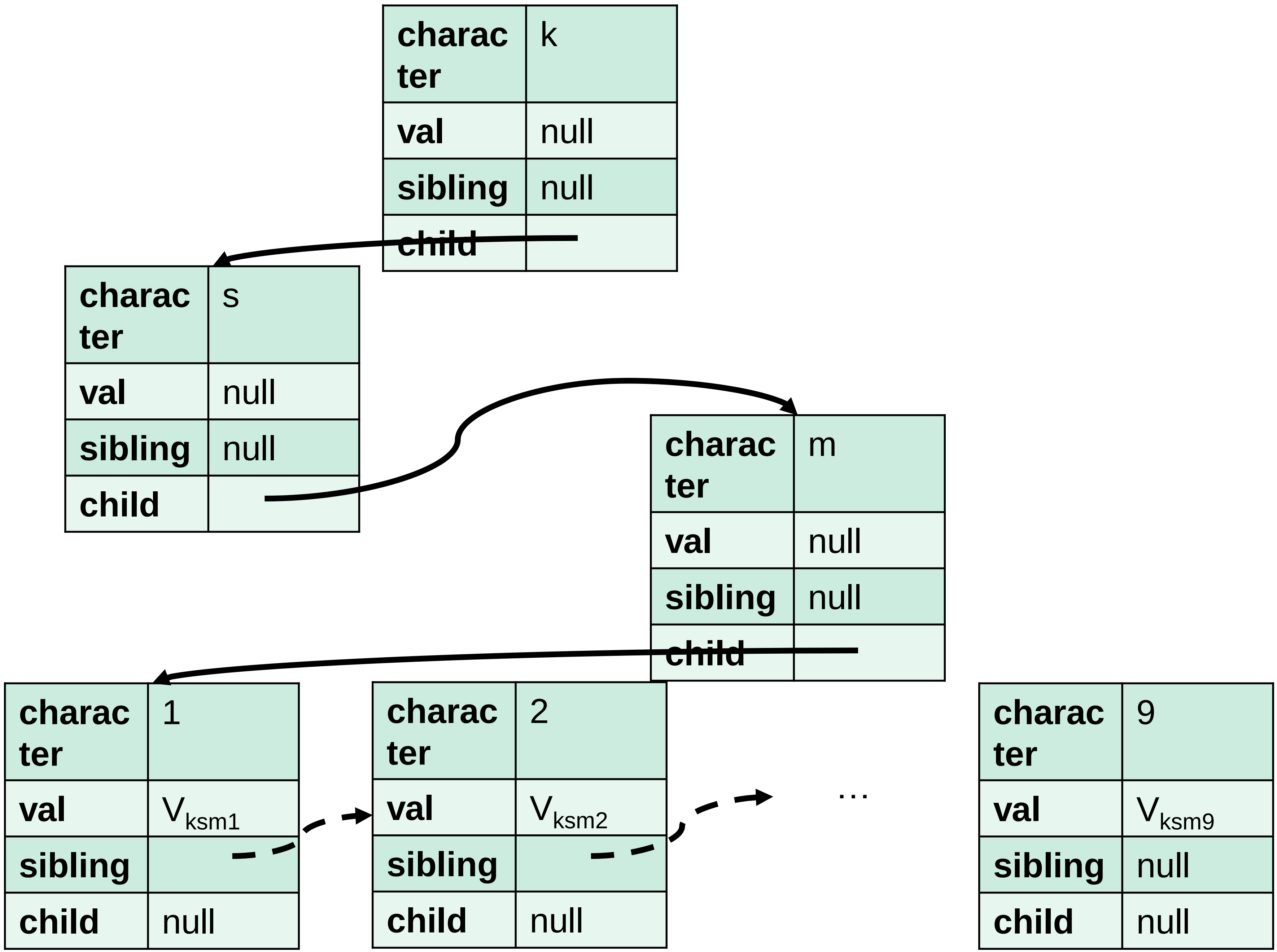
Total space is 13*1028 = 13,364 bytes

DLB Trie

```
private class DLBNode<T> {
    private Character character;
    private Object val;
    private Node sibling;
    private Node child;
}
```

Each node takes 4*4 = 16 bytes, assuming 4 bytes per reference variable

DLB Trie



character k
val null
sibling null
child
character s
val null
sibling null
child
character m
val null
sibling null
child
character 1
val V_{ksm1}
sibling
child null
character 2
val V_{ksm2}
sibling
child null
…
character 9
val V_{ksm9}
sibling null
child null

DLB Trie

We will end up with 12 nodes

Total space is 12*16 = 192 bytes

Compare to 13,364 bytes with an R-way RST

Runtime Comparison for Search Trees/Tries

	Search hit	Search miss (average)	insert
BST	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
RB-BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
DST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
RST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
R-way RST	$\Theta(w)$	$\Theta(\log_R n)$	$\Theta(w)$
DLB	$\Theta(wR)$	$\Theta(\log_R n \cdot R)$	$\Theta(w \cdot R)$

Final notes on Search Tree/Tries

- We did not present an exhaustive look at search trees/tries, just the sampling that we're going to focus on
- Many variations on these techniques exist and perform quite well in different circumstances
  - Ternary search Tries
  - R-way tries without 1-way branching
- See the table at the end of Section 5.2 of the text